Clube Delphi

Delphi 3D
Ambientes 3D de maneira fácil no Delphi

Quick Update
Da VCL para o FireMonkey

Ano 12 :: Edição 136 :: R$ 11,90

DevMedia

ISSN 1517990-7

FireMonkey + DBExpress

Aprenda a revitalizar seus aplicativos

Mini-Curso sobre Banco de Dados
Firebird –
Crie e manipule banco de dados

Boas Práticas – Orientação a Objetos
Aprofundando no uso de Interfaces

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Sumário

Delphi

Quick Update

06 – Da VCL para o FireMonkey
[*Paulo Quicoli*]

Delphi

10 – Delphi 3D – Utilizando OpenGL
[*Giuliano Scombatti Pinto*]

Mini-Curso/ Banco de Dados

18 – Firebird: Poderoso, Leve, Gratuito e Open-Source
[*Fabiano Moura*]

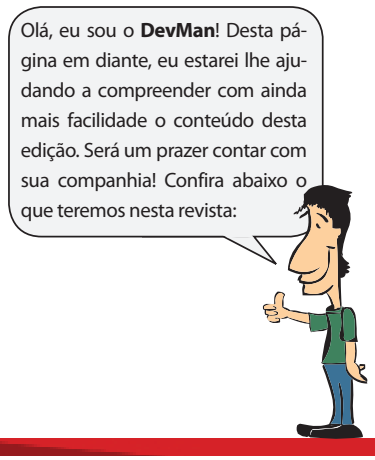
Delphi/ Novidades

24 – FireMonkey, desenvolvimento com banco de dados
[*Fabrício Hissao Kawata*]

Boas Práticas

Boas Práticas

36 – Aprofundando em Interfaces
[*Vitor Luiz Rubio*]



Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Clube Delphi

Ano 12 - 136ª Edição 2011 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@devmedia.com.br

Editor Técnico

Leandro Daniel
contato@leandrodaniel.com

Equipe Editorial

Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Guinther Pauli
guintherpauli@gmail.com

Jornalista Responsável

Kaline Dolabella - JP24185

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/mancad
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

EDITORIAL

O FireMonkey está aí e veio para ficar. Em seu site de apresentação na Embacadero diz que esta é plataforma para a próxima geração de aplicativos de negócio. Eu acredito nisso. Cada vez mais a interface gráfica tem ganhado atenção, não só dos fabricantes, mas dos próprios usuários. Ela tem definido o diferencial entre um aplicativo e outro em uma negociação comercial.

Prova disso é o futuro Windows 8 que teve sua interface remodelada para atingir um público que não se contenta mais apenas com um mouse, mas um público que está em movimento e interage com seus corpo, mais especificadamente, seus dedos. O desenvolvedores Delphi tem à sua disposição esse novo framework que possibilita a criação de interfaces riquíssimas que aliadas à velocidade dos aplicativos Delphi farão seus negócios alavancar. Mas por onde começar?

Essa é a dúvida de muitos e pensando nisso, o Fabrício apresenta um artigo interessante que mostra como fazer um aplicativo FireMonkey que acessa banco de dados e ainda mostra como personalizar essa aplicação. Ainda no quesito visual o Giuliano mostra como fazer uso da OpenGL em aplicativos Delphi para se criar ambientes 3D. Utilizando componentes que auxiliam nessa tarefa é possível construir animações e até mesmo jogos, este é um mercado que pode ser muito explorado.

Mas do que adianta ter uma "carinha bonita" se o resultado não é o esperado? Por isso temos a engenharia de software, para dar consistência ao que realmente é importante. Assim, o Vitor mostra em seu artigo como implementar e utilizar de forma eficiente interfaces. Este é um recurso importantíssimo da orientação a objetos. No livro Design Patterns Explained: A New Perspective on Object-Oriented Design é dito que um dos princípios para se alcançar sistema flexível é encapsular aquilo que varia na sua regra de negócio, encapsular em interfaces e programar sobre as mesmas. É justamente isso que é mostrado no artigo.

E para encerrar a edição, o Fabiano mostra como utilizar o Firebird utilizando seus aplicativos de linha de comando próprios, aqueles que nunca usamos. Por exemplo, se você precisar fazer algum ajuste em algum cliente e você não tem à disposição o IBExpert (cito o IBExpert por sua popularidade), o que fazer? Você pode utilizar o utilitário ISQL e via prompt realizar suas tarefas. E nesse mesmo artigo, se você leitor é iniciante no assunto, entenderá como criar um banco de dados, tabelas, chaves e como manipular informações.

Boa leitura!

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

Assinatura
ClubeDelphi PLUS

DevMedia

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 1,20

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Da VCL para o FireMo

Toda ferramenta para evoluir precisa quebrar seus laços com o passado. Com o Delphi não foi diferente. Estávamos a alguns anos recebendo novas versões do Delphi, porém sempre parecia "tudo do mesmo". Eu acredito que ao analisar o mercado a Embarcadero sentiu a necessidade mudar sua estratégia de compatibilidade para trazer algo realmente novo aos desenvolvedores. E isso aconteceu.

Surge então o FireMonkey com o foco de criar aplicações de negócio visualmente ricas. Veja, não é para criação de joguinhos e sim aplicativos de negócio.

Essa notícia trouxe grande entusiasmo aos desenvolvedores. O FireMonkey possui uma API própria agnóstica de sistema operacional, por isso ele foi escolhido pela Embarcadero como plataforma visual para desenvolvimento multiplataforma. Nele os controles são desenhados por sua engine, o que permite ir além das APIs de qualquer SO existente. Estão disponíveis animações, efeitos visuais, temas e muito mais. Sua aplicação não será mais a mesma visualmente.

Estrutura de Classes

Vamos fazer um breve comparativo entre a VCL e o FireMonkey. Ambos possuem uma mesma hierarquia inicial. Começam com a classe TObject, então continuam com TPersistent e TComponent. A partir daqui tudo fica diferente. Na **Figura 1** temos as classes que são o fundamento da VCL.

Agora vamos olhar na **Figura 2** e admirar a estrutura do FireMonkey.

Como se pode notar o FireMonkey já foi planejado em sua estrutura para ser um framework de riqueza visual. Temos classes que representam estilos e controles que permitem ser estilizados e até mesmo estruturalmente ser modificados. Na VCL não temos isso, qualquer estilização é feita através de eventos do sistema operacional que permitem o uso de um desenho diferenciado, a VCL atua utilizando 100% das APIs Windows para exibição de informações.

No FireMonkey esse desenho não depende do sistema operacional. Agora que conseguimos visualizar as diferenças fica a pergunta: como posso utilizar essa riqueza visual em meu aplicativo existente, VCL, que está rodando estavelmente?

A resposta oficial é simples, você não pode. Não existe compatibilidade entre FireMonkey e VCL. Você não pode inserir janelas ou controles em um aplicativo VCL Forms. A natureza do FireMonkey difere muito além de suas classes, sua principal diferença é conceitual, essa foi a quebra de paradigma da Embarcadero.

nkey

Paulo Quicoli

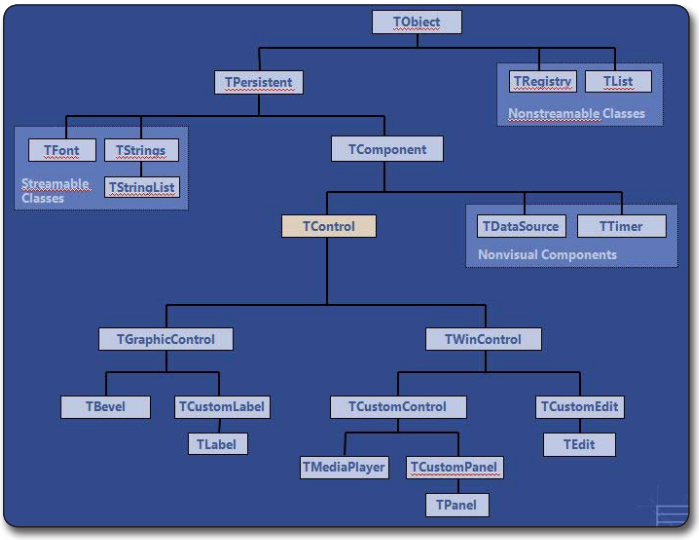


Figura 1. Hierarquia base da VCL

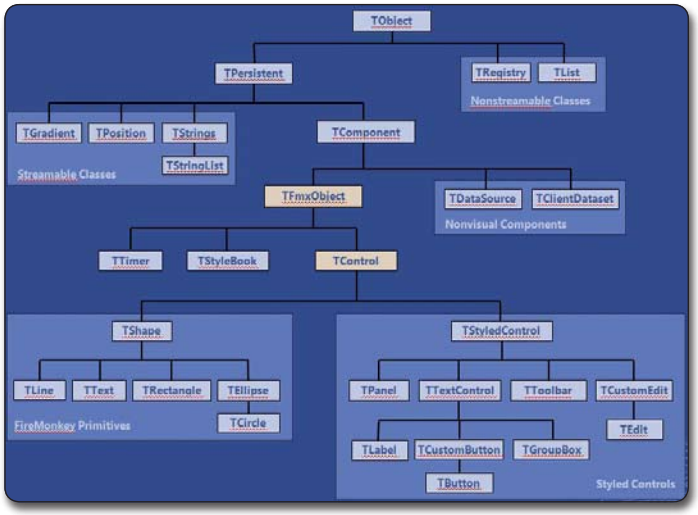


Figura 2. Hierarquia base do FireMonkey

A mudança para o FireMonkey não é apenas uma troca de componente, existe uma curva de aprendizado, novos conceitos, novas classes, novas convenções e comportamentos. A posição da Embarcadero quando ao relacionamento VCL x FireMonkey é clara:

• A Embarcadero não suporta oficialmente o uso de formulários FireMonkey em projetos VCL;

• Você não pode utilizar controles FireMonkey em formulários VCL, as duas plataformas diferem muito entre si para que isso aconteça;

• O FireMonkey trabalha de uma forma muito diferente em relação à VCL, então, qualquer comunicação intra-formulário entre controles/formulários VCL e FireMonkey poderia ser problemática.

Mas existe uma saída, a não oficial. Existem dois projetos que podem ajudar o desenvolver atual a resolver essa questão. Basicamente temos duas soluções: converter ou integrar as tecnologias.

Mida

Este é um projeto que propõe algo que vai além da conversão da VCL para FireMonkey. Segundo o autor da ferramenta ela não é apenas um conversor de componentes, mas um conversor de projetos. Atualmente está preparada mais de 200 componentes, incluindo os nativos da VCL e de terceiros. A conversão de DataModules, código fonte e projetos (.dpr) acontece de forma automática. Ao final desse trabalho um log completo do que foi feito é gerado, incluindo dicas para auxiliar na conversão do que não foi convertido plenamente. Uma aplicação comercial tradicional que utiliza controles dbaware pode ser transformada completamente em uma aplicação FireMonkey, até mesmo sua forma de ligação de dados é atualizada para o uso do LiveBinding, ou seja, todo o suporte a banco de dados existente é mantido e atualizado.

O roadmap dessa aplicação Mida é que em breve sejam disponibilizados pontos de extensão que permitam às empresas que desenvolvem componentes VCL criar uma espécie de transformador de componentes, assim, qualquer componente VCL pode ser transformado pelo próprio fabricante. Além disso, o apoio para a conversão do código fonte de todo um projeto será aprimorado para incluir dicas significativas quando algo não compatível for encontrado. Usando esta ferramenta talvez o que levaria semanas ou meses pode levar dias. Toda essa ajuda tem um custo, a ferramenta não é gratuita, contudo, se compararmos seu valor com o valor das horas de desenvolvimento que seriam utilizadas para fazer a migração manualmente, seu preço é vantajoso.

MonkeyGroomer e MonkeyMixer

Existe outro projeto chamado MonkeyGroomer, que é gratuito, e faz um trabalho similar ao Mida, porém tem seu foco inicial mais voltado a converter os controles VGScene para o formato FireMonkey. Na realidade eles são a "origem" do FireMonkey. A empresa que desenvolvia esses componentes foi adquirida pela Embarcadero que então utilizou essa base de conhecimento para aprimorar o produto e criar o FireMonkey.

Se você não quer converter nada, mas gostaria de dar um upgrade visual em algo que você já tem, esta é a ferramenta. Apesar do não apoio da Embarcadero o MonkeyMixer permite que você utilize o FireMonkey em aplicações VCL. Na realidade os autores das duas ferramentas estão se unindo para criar um único plugin que unirá as duas funcionalidades. Atualmente o MonkeyGroomer utiliza o conceito de modelos de script de conversão. Esses modelos fornecem à engine do MonkeyGroomer toda informação necessária sobre os tipos VCL e FireMonkey de algo que será convertido mais as instruções para a conversão. Esses scripts podem se auto atualizar a partir de um repositório online e nesse conceito os próprios usuários podem criar os seus próprios. Além disso, não são apenas os formulários VCL (arquivos DFM) que são analisados e quebrados em arquivos FMX do FireMonkey, os arquivos PAS associados também são modificados, incluindo os arquivos DPR e DPROJ. Caso algo não possa ser convertido, a ferramenta tenta localizar no repositório online um script que solucione o problema, se não encontrar isso é registrado em um relatório ao final da conversão.

Conclusão

A mudança para o FireMonkey é inevitável para quem quer continuar competitivo no mercado. Três ferramentas foram expostas para acelerar essa adoção e até mesmo auxiliar no aprendizado, agora é com vocês.

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas e coordenador técnico da equipe de desenvolvimento .NET da Siplan Control-M (www.siplancontrolm.com.br), e professor da FATEC, campus Taquaritinga. Blog: http://quicoli.worpress.com

MonkeyGroomer
http://pascalcoder.blogspot.com/

MonkeyMixer
http://www.simonjstuart.com/2011/10/19/world-meet-monkeymixer-use-firemonkey-forms-directly-in-vcl-projects/

Mida
http://midafiremonkey.wordpress.com/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

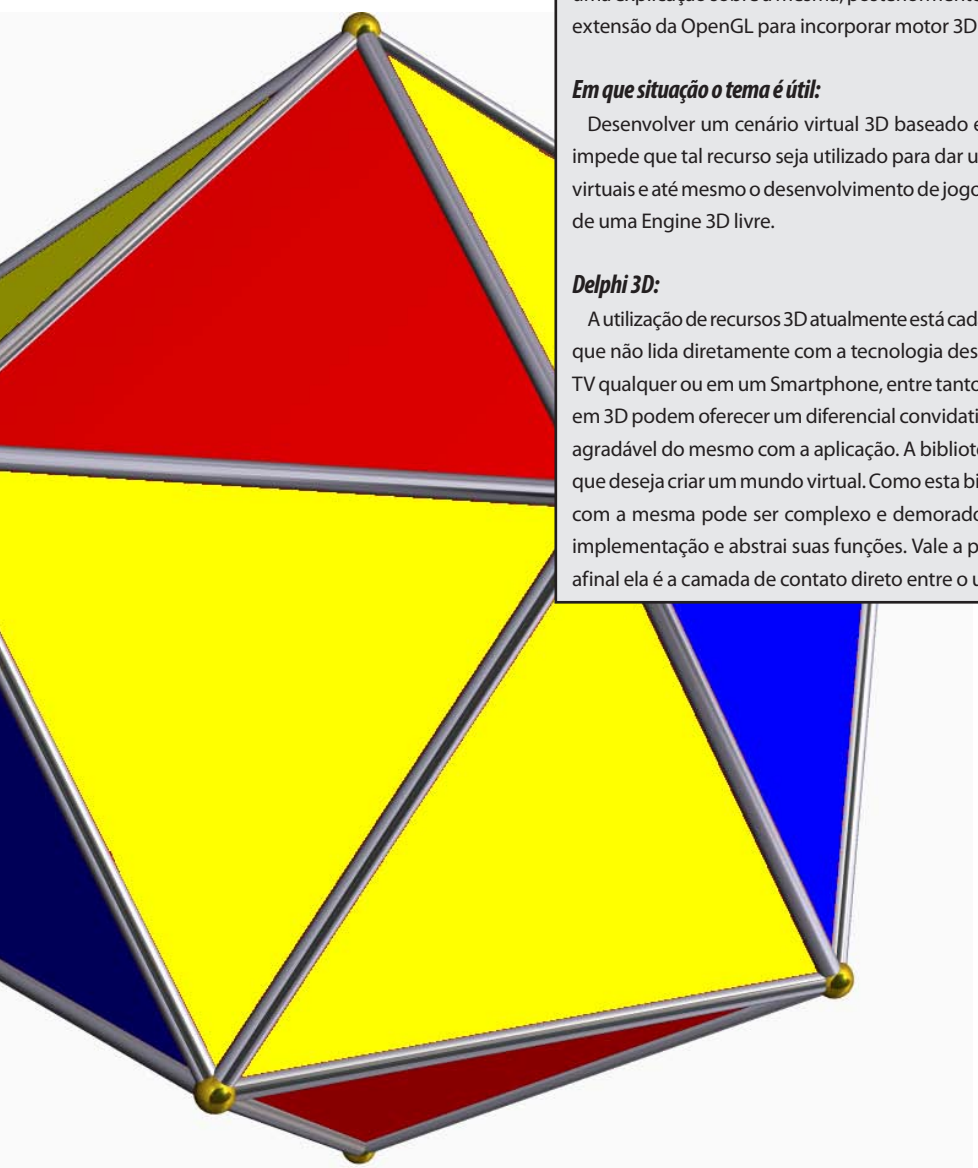
Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

Delphi 3D

Utilizando OpenGL em Aplicaç



Resumo DevMan

De que trata o artigo:

Este artigo demonstra como utilizar os recursos presentes na OpenGL, uma biblioteca gráfica portável presente em vários sistemas operacionais, através do Delphi. Em um primeiro momento é apresentada uma explicação sobre a mesma, posteriormente sua implementação através da biblioteca GLScene, uma extensão da OpenGL para incorporar motor 3D na aplicação VCL.

Em que situação o tema é útil:

Desenvolver um cenário virtual 3D baseado em OpenGL para ser utilizado na aplicação, porém nada impede que tal recurso seja utilizado para dar um pontapé inicial na composição de complexos mundos virtuais e até mesmo o desenvolvimento de jogos para PC, uma vez que neste artigo existe a apresentação de uma Engine 3D livre.

Delphi 3D:

A utilização de recursos 3D atualmente está cada vez mais presente na vida das pessoas. Até mesmo aquele que não lida diretamente com a tecnologia desfruta deste artifício, seja no cinema, em um programa de TV qualquer ou em um Smartphone, entre tantos outros dispositivos eletrônicos. Os ambientes baseados em 3D podem oferecer um diferencial convidativo ao usuário, bem como propiciar uma experiência mais agradável do mesmo com a aplicação. A biblioteca OpenGL é uma boa alternativa para o desenvolvedor que deseja criar um mundo virtual. Como esta biblioteca possui suas particularidades, o desenvolvimento com a mesma pode ser complexo e demorado, entretanto basta utilizar a GLScene que simplifica sua implementação e abstrai suas funções. Vale a pena investir na aparência com a utilização deste recurso, afinal ela é a camada de contato direto entre o usuário e a aplicação.

A computação gráfica ganha cada vez mais espaço na atualidade. A criação e utilização de ambientes virtuais 3D e seus objetos é cada vez mais empregada no cinema, nos jogos e até mesmo nas aplicações de usuário. Tais ambientes envolvem uma diversidade de cálculos, figuras, texturas e muita criatividade, uma vez que o desenvolvedor não possui limites de software para a criação de cenas, personagens, entre outros. É evidente que existem várias ferramentas para que o objetivo final seja alcançado pelo desenvolvedor, que podem ser utilizadas individualmente ou em conjunto, de acordo com a necessidade do projeto. O Delphi também pode ser

Giuliano Scombatti Pinto

empregado para desenvolver ambientes 3D, cenas, objetos e efeitos utilizando a própria VCL. Embora seu foco não seja tal desenvolvimento, é possível utilizar APIs e Componentes para tanto, como a GLScene que será descrita posteriormente neste artigo, assim como importar objetos gráficos desenvolvidos em outras aplicações para o projeto VCL, como é o caso de modelos 3D criados com o Software Blender. Entretanto, para os desenvolvedores que possuem a versão XE 2 do Delphi, a biblioteca Firemonkey pode ser utilizada para a elaboração de animações e efeitos 3D na aplicação sem que exista a necessidade de adicionar objetos de terceiros.

A OpenGL

A OpenGL é uma biblioteca de baixo-nível que foi desenvolvida inicialmente pela Silicon Graphis Inc. que possui diversas rotinas gráficas para utilização em ambientes 2D e 3D, possuindo alto desempenho e fácil portabilidade. Por suas características, a OpenGL não contém funções específicas para gerenciar janelas, utilizar arquivos de entrada/saída e interagir com o usuário, uma vez que tais funções estão presentes no ambiente do Sistema Operacional. Ela permite a utilização de figuras primitivas (linhas, triângulos, cubos, polígonos, etc) e efeitos como mapeamento de texturas, iluminação, transparência e colorização para o desenvolvimento de ferramentas CAD, mundos virtuais, animações e cenas em uma aplicação. Ela é baseada em fundamentos matemáticos, possuindo um sistema de coordenadas 3D (Eixos X, Y e Z). Ao utilizar a OpenGL o desenvolvedor define as dimensões do ambiente virtual 3D, bem como o ponto de vista da cena. Com relação aos objetos que serão utilizados na cena, uma vez que a OpenGL suporta apenas as formas primitivas citadas anteriormente, se o objetivo do desenvolvedor for desenhar uma esfera, que é um objeto gráfico mais complexo, o mesmo poderá utilizar vários polígonos em conjunto, como triângulos, tornando a esfera praticamente perfeita. Esta regra também é valida para qualquer desenho de alto nível com a OpenGL.

Os comandos e subframeworks

Ao desenhar utilizando diretamente a OpenGL, o desenvolvedor informa o início e o fim de um determinado desenho através dos comandos glBegin() e glEnd(). Com relação aos nomes das funções da OpenGL, existe um padrão para facilitar na utilização da mesma.

Normalmente os nomes das funções são constituídos pelo prefixo "gl", que representa a própria biblioteca, a descrição da função ao centro e um sufixo, que indica que a função precisa de determinado número de parâmetros, bem como seus tipos. Sendo assim, as funções são compostas por gl-[descrição]-[quantidade-argumentos]-[tipo-argumentos].

A OpenGL também é constituída por uma série de variáveis de estado que indicam os estilos, as posições, os tipos de luzes, entre outros atributos que podem ser alterados. Tal alteração pode ser executada através dos comandos glEnable(), para habilitar determinado estado e glDisable(), para desabilitar. Por exemplo, para habilitar a luz em determinado ambiente basta utilizar glEnable(GL_LIGHTING). Também é possível saber se determinado estado esta habilitado através de glIsEnabled().

No ambiente Windows a biblioteca gráfica OpenGL está disponível normalmente através das DLLs opengl32.dll e glu32.dll que são instaladas automaticamente com o Windows e estão presentes no diretório System32. Para sua utilização em uma aplicação Delphi, sem a necessidade da obtenção de componentes específicos, pode-se incluir na seção Uses a Unit Winapi.OpenGL ou simplesmente OpenGL. Entretanto existem outras Units desenvolvidas por terceiros e livres, que também implementam as funções da OpenGL e atribuem algumas particularidades, como é o caso da dglOpenGL.

Para a visualização de uma cena baseada nesta biblioteca gráfica, o desenvolvedor pode atribuir a renderização a um dispositivo de contexto, como um TForm ou um TPanel, informando o handle (manipulador) de tal objeto e utilizar algumas funções para a inicialização.

Através da OpenGL, o desenvolvedor também consegue manipular objetos em uma cena, em outras palavras, ele pode rotacionar, mover e alterar o mesmo. Para criar uma animação, por exemplo, basta executar os comandos na ordem em que se deseja obter a cena. É importante frisar que a OpenGL utiliza matrizes para executar as transformações geométricas desejadas. Portanto, a cada nova transformação a matriz é alterada e utilizada para o desenho dos objetos a partir do atual momento, fazendo uma composição de matrizes. As funções glPushMatrix() utilizada para salvar a atual matriz em uma pilha da OpenGL, e glPopMatrix() utilizada para restaurar, devem ser utilizadas para evitar um efeito cumulativo.

Com relação à execução de uma animação com a biblioteca OpenGL é importante destacar a utilização de Double-buffering. Através do Double-buffering um desenho é mostrado na tela enquanto é realizada a renderização de outro, em uma Offscreen-buffer, evidentemente porque alguns desenhos podem ser complexos e consequentemente levarem mais tempo para serem elaborados, criando

um efeito indesejado onde cada parte que constitui o desenho é mostrada parcialmente, e através do buffer, o objeto só será exibido quando estiver completamente criado. Em uma animação, por exemplo, cada frame (quadro) é criado no Offscreen-buffer primeiramente e só será exibido depois que estiver pronto. Para concluir, também é possível gerenciar eventos do teclado e do mouse com a OpenGL, criando uma interação entro o usuário.

A biblioteca GLScene

Como já mencionado anteriormente, a OpenGL é uma biblioteca gráfica padrão e portável, entretanto existem implementações específicas para ela em vários ambientes como Linux e Windows por exemplo. Sendo assim, existem componentes destinados a diversas linguagens de programação, como é o caso do Visual C++ e do Delphi, que simplificam seu uso e aumentam a produtividade. No caso do Delphi uma das bibliotecas baseadas em OpenGL mais conhecida é a GLScene, sendo gratuita e licenciada sob a Mozilla Public Licence (Licença Pública Mozilla). Ela possui vários componentes e objetos que permitem a criação e a renderização de cenas 3D com a OpenGL de uma maneira mais fácil e rápida, sem maiores dificuldades, mantendo o poder gráfico da biblioteca. Vale a pena evidenciar que a GLScene não é simplesmente uma extensão ou um utilitário da OpenGL, ela tem evoluído e se tornado uma Engine 3D (Motor gráfico), mantendo também o foco no desenvolvimento rápido da aplicação.

Através da GLScene o desenvolvedor consegue desenhar e renderizar suas cenas e efeitos sem que o mesmo tenha conhecimentos específicos de OpenGL, sem contar que ao obter a biblioteca o desenvolvedor conta com uma diversidade de exemplos que oferecem uma base e demonstram a facilidade do uso da mesma.

A GLScene possui uma estrutura de objetos hierárquica expansível, permitindo que o desenvolvedor preencha a cena com quantos objetos desejar ou até que a memória suporte. Ela permite o gerenciamento das cenas de uma maneira interativa, onde o uso de efeitos de movimentação e rotação se torna muito simples, bem como a adição de câmera e luz. Possui suporte a sistemas de partículas, modelos 3D criados em outras ferramentas, vários formatos de arquivos, texturas entre outros, simplificando também a utilização de materiais. Se o desenvolvedor preferir, também pode utilizar classes de suporte desta biblioteca para obter acesso direto à OpenGL.

Quanto a questão da renderização (Rendering), a GLScene usa o driver de hardware OpenGL de uma maneira automática, também permitindo a visualização à partir de vários pontos da cena, bem

Nota do DevMan

O handle é um identificador utilizado pelo Windows para se referir a um dispositivo de contexto, como uma janela. Sendo assim, sempre que o desenvolvedor desejar se referir a determinado dispositivo pode informar seu handle, uma vez que cada dispositivo possui um identificador único.Em outras palavras, um controle derivado de um TWinControl possui um handle para que possa ser identificado e acessado pelo sistema operacional Windows.

Nota do DevMan

Ferramentas CAD (Computer Aided Design) são aplicações utilizadas para elaboração de desenhos através do computador. Normalmente são empregadas na criação de desenhos técnicos bidimensionais ou tridimensionais, permitindo que desenhistas, engenheiros, arquitetos, dentre outros profissionais, utilizem seus recursos para o desenvolvimento dos seus projetos.

Nota do DevMan

Para simplificar a programação com a OpenGL foram desenvolvidas algumas bibliotecas como a GLU, GLUT, GLX e FSG. A GLU (OpenGL Utility Library) é composta por uma coleção de rotinas que utilizam as funções de baixo nível da OpenGL para a definição de matrizes para projeção, bem como renderização de superfícies e visualização. Já a GLUT (OpenGL Utility Toolkit) oferece várias ferramentas independentes da plataforma, incluindo elementos como menus Pop-up e suporte a joystick, abstraindo também a complexidade de alguns códigos. A GLX (OpenGL Extension to the X Window System) é destinada as máquinas que utilizam o X Window System. No caso do Windows, são utilizadas as funções WGL para fornecer janelas para a OpenGL. A FSG (Fahrenheit Scene Graph) é um conjunto de ferramentas orientadas a objetos com base na OpenGL, para a elaboração de softwares 3D interativos. A utilização destas bibliotecas depende do sistema operacional onde a aplicação será executada.

Nota do DevMan

O X Window System (ou simplemente X) é um sistema que permite a utilização de uma interface gráfica com o emprego de janelas e controles para sistemas Unix Like, como é o caso do Linux, porém pode ser encontrado em outras plataformas também.

Nota do DevMan

A Licença Pública Mozilla define que o código fonte do software é aberto, em outras palavras, software livre, e pode ser obtido com todos seus módulos (ou parte deles) para que o desenvolvedor possa alterar, corrigir, excluir, estudar, etc. Entretanto, quando tal código for alterado e copiado por um terceiro, o mesmo deve permanecer sob tal licença.

Nota do DevMan

A renderização consiste na conversão de diversos objetos gráficos em um ambiente visual, onde são aplicados efeitos de luz, textura, transparência, cor, sombra e reflexão a tais objetos, formando uma cena 3D por exemplo, onde posteriormente é efetuada uma representação da cena em 2D. Este é um recurso muito utilizado na computação gráfica.

Nota do DevMan

O Bitmap (BMP) é um formato de arquivo utilizado para o armazenamento de imagens digitais de diversas dimensões e qualidade. É constituído por um conjunto de pixels onde cada um armazena uma cor como se fosse uma célula em uma tabela com várias linhas e colunas de acordo com a dimensão da imagem. É muito utilizado na computação gráfica para aplicação de texturas a objetos e está presente em vários sistemas operacionais.

como exportando a cena direto para um arquivo de Bitmap ou até mesmo para impressora. Oferece suporte ao modo tela-cheia, alteração de resoluções, reflexões, efeitos de espelho, sombras entre muitos outros.

Com relação à animação, a GLScene permite a progressão de uma cena através do tempo. Também possuí suporte a física dinâmica como inércia, aplicação de força, umidade e aceleração.

Para concluir, a GLScene também possui suporte a som, vários utilitários e integração com outras bibliotecas desenvolvidas por terceiros.

Utilizando a GLScene

A GLScene pode ser adicionada facilmente ao Delphi como um componente (ver na seção links o endereço para obtenção da biblioteca). Partindo do princípio que a biblioteca está devidamente instalada, para a criação de uma cena básica com a GLScene basta adicionar um componente do tipo TGLScene e um do tipo TGLSceneViewer. O TGLSceneViwer é um componente visual que exibe uma janela de renderização (visualização do mundo virtual 3D), que pode ser redimensionado conforme o desenvolvedor desejar. Uma boa prática é alterar a propriedade Align deste componente para alCliente, fazendo com que o mesmo ocupe o tamanho disponível no formulário. Para editar uma cena, basta dar um clique duplo no componente TGLScene que a janela GLScene Editor será exibida. Através desta janela o desenvolvedor pode adicionar vários componentes na cena, como uma câmera, um efeito de luz, um objeto qualquer e outros. Para tanto, basta clicar com o botão direito em Scene objects, dentro de Scene root, e escolher o objeto que se deseja adionar em Add object, que possui uma variedade de submenus com várias opções. Vale a pena evidenciar que os objetos são colocados na coordenada xyz [0,0,0]. Sendo assim, para posicionar os objetos da maneira desejada na cena, bem como definir seu tamanho, é importante que o desenvolvedor tenha noções dos eixos X, Y e Z. Lembrando que no sistema de coordenadas, a reta X representa a horizontal, a reta Y a vertical, e a reta Z a profundidade.

Para a visualização da cena a propriedade TargetObject da câmera deve ser configurada para o objeto inserido. A luz também deve ser ajustada com relação à câmera.

Vale a pena evidenciar que, quanto aos objetos da GLScene, as propriedades devem sempre ser alteradas, para que se obtenha o efeito desejado, bem como a ligação com outros objetos. A **Tabela 1** mostra as propriedades mais comuns dos objetos da GLScene.

Quanto à questão dos Materiais, a propriedade Materials dos objetos e suas sub-propriedades são muito importantes para a definição das faces dos objetos. Através dela é possível mudar a cor do objeto selecionado acessando e alterando a sub-propriedade FrontProperties.Diffuse. Também é possível definir a transparência da cor através da propriedade Alpha, preenchendo valores entre 0 e 1, porém a propriedade BlendingMode deste Material deve ser configurada com o valor bmTransparency para tanto. Além de alterar as cores, através de Materiais, o desenvolvedor pode adicionar texturas aos objetos, permitindo assim atribuir uma imagem ao mesmo e

Propriedade	Descrição
Behaviors	Utilizado para adicionar um comportamento a um objeto. Normalmente é destinado a alterar o movimento, a colisão e o som dos mesmos.
Children	Permite acessar a lista de objetos filhos que pertencem a certo objeto.
Direction	Vetor utilizado para definir a 'frente' do objeto.
Effects	Permite adicionar efeitos ao objeto.
Objects Sorting	Ordem de renderização dos objetos
Material	Utilizado para adicionar um material ao objeto, como exemplo uma imagem.
Name	Seu nome.
Parent	Permite acessar o 'Pai' do Objeto.
PitchAngle	Define a rotação do objeto com relação ao eixo Y (Valor em graus)
Position	Vetor utilizado para definir a posição do Objeto no cenário 3D.
RollAngle	Define a rotação do objeto com relação ao eixo X (Valor em graus).
Scale	Vetor que define o tamanho do objeto.
ShowAxes	Permite ativar ou desativar a exibição dos eixos através de linhas coloridas.
TagFloat	Assemelha-se à propriedade Tag.
TurnAngle	Define a rotação do objeto com relação ao eixo Z (Valor em graus).
Up	Vetor utilizado para definir qual é a área superior do objeto.
Visibility culling	Utilizada para acelerar a renderização do objeto.
Visible	Define se o objeto será exibido.

Tabela 1. Propriedades comuns dos Objetos da GLScene

fazendo com que ele fique com uma aparência mais real. Para tanto basta acessar a sub-propriedade Texture, presente em Materials. Através dela, além de adicionar a imagem desejada, o desenvolvedor também define como ela será aplicada, lembrando também que sua sub-propriedade Disabled deve conter o valor False. É importante que a imagem utilizada seja distribuída juntamente com a aplicação. Também é possível utilizar o recurso Material Editor, presente no botão adicional da propriedade Materials, perimitindo assim a edição de materiais e texturas de uma maneira mais simples.

A GLScene também possui um componente destinado ao armazenamento de materiais, o TGLMaterialLibrary, uma biblioteca para armazenar os mesmos e permitir que sejam aplicados a vários objetos, uma vez que uma mesma textura pode ser utilizada em vários componentes da cena. Entretanto, o desenvolvedor pode também carregar a textura de um objeto de uma maneira dinâmica, utilizando a função [Objeto].Material.Texture.Image.LoadFromFile().

O Componente TGLCadencer

O TGLCadencer é mais um componente presente na GLScene. Sua função é bem similar ao tradicional Timer já contido no Delphi,

entretanto os eventos não ocorrem em um tempo pré-definido, mas após a renderização de um quadro. Vale a pena evidenciar que o trecho de código que se deseja executar nesta progressão deve ser colocado no evento OnProgress, sendo também interessante sincronizar esses eventos com o TGLSceneViwer, gerando uma nova renderização através do comando GLSceneViewer1.Invalidate. Para obter a taxa de quadros por segundo em que os objetos estão sendo renderizados, basta utilizar a função FramesPerSecondText, e posteriormente para mantê-la atualizada a função ResetPerformanceMonitor, ambas do TGLSceneViewer.

Caso o desenvolvedor deseje utilizar o tempo para realizar os eventos de progressão, ele pode contar com as funções newTime e deltaTime. A função newTime permite a obtenção do tempo atual e a deltaTime representa o intervalo originado do último evento de progressão realizado. Para que o TGLCadence, funcione também é necessário que ele seja interligado a cena. Para tanto, basta configurar sua propriedade Scene para a cena desejada.

Interagindo com a cena

A GLScene permite de uma maneira bem simplificada, através do uso de componentes e pequenos trechos de código, que o usuário interaja com a aplicação através do teclado ou do mouse por exemplo. Para tanto, o componente TGLNavigator pode ser adicionado ao projeto para suporte a tais manipulações. A propriedade MovingObject deste componente deve ser alterada para a câmera ou objeto a qual se deseja interagir. Também se pode utilizar um TGLCadencer para realizar uma verificação periódica do estado das teclas e adicionar a Unit GLKeyborard ao projeto para a utilização das funções de teclado.

O mouse também pode ser utilizado para interação, adicionando um componente TGLUserInterface. Ele permite que o usuário gire livremente a cena ou o objeto selecionado ao movimentar o mouse. Para sua utilização, deve-se ajustar as propriedades GLNavigator e GLVerticalNavigator, informando o objeto TGLNavigator que deverá ser utilizado. A velocidade de movimento também dever ser configurada na propriedade MouseSpeed. Para que o mouse funcione corretamente também é necessário adicionar a função GLUserInterface1.MouseLookActive no evento OnCreate do Formulário, para que o cursor do mouse seja capturado pelo navegador, bem como adicionar as funções GLUserInterface1.MouseUpdate e GLUserInterface1.MouseLook no GLCadencer, para que a visão obtida com os movimentos do mouse seja atualizada.

Criando uma simples animação

A GLScene será utilizada para a criação de uma simples animação 3D contendo um cubo com efeito de rotação. Como visto anteriormente, será necessário adicionar os componentes básicos da cena desta biblioteca, o TGLScene e o TGLSceneViewer no formulário. Depois de adicionados, a propriedade Align do GLSceneViewer1 deve ser alterada para alClient. O próximo passo é dar um clique duplo no Objeto GLScene para abrir o editor da cena. Em Scene objects basta dar um clique com o botão direito e adicionar uma câmera (Add camera). Novamente, um clique com o botão direito em Scene objects e na opção Add Object, acessar o submenu Basic geometry e selecionar Cube, para a adição do cubo. Agora deve-se clicar no objeto GLCamera1 e ajustar as coordenadas da propriedade Position para definir o ponto de vista do observador com relação a cena. Pode-se ajustar Y = 3 para conseguir visualizar o cubo.

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Também deve ser alterada a propriedade TargetObject que indica qual o Objeto que a câmera está focada, para GLCube1, o cubo adicionado anteriormente. O próximo passo é selecionar o objeto GLCube1 e na propriedade Position, alterar as coordenadas para X = 2, Y = 2 e Z = 2, lembrando que o objeto é colocado nas coordenadas [0,0,0]. Ao selecionar o objeto GLSceneViewer1 alteramos a propriedade Camera para GLCamera1. A partir deste momento já será possível visualizar em 3D o cubo que foi adicionado.

Dando sequência pode-se aplicar um movimento de rotação ao cubo adicionando um objeto do tipo TGLCadencer. Depois de adicionado deve-se selecionar o objeto GLCadencer1 e alterar a propriedade Scene para GLScene1. Posteriormente, deve-se dar um clique duplo no objeto para criar o evento OnProgress. Como o objetivo é criar um efeito de rotação do cubo pode se alterar a propriedade TurnAngle do GLCube1 para 90º multiplicado pelo tempo atual, através de newTime. Também se pode exibir a quantidade de quadros por segundo (FPS) da animação em um recipiente visual qualquer, como na propriedade Caption do formulário através de FramesPerSecondText do objeto GLSceneViewer.

Quanto à interação da aplicação com o usuário através do teclado e mouse, deve-se adicionar um componente do tipo TGLNavigator e um do tipo TGLUserInterface. A propriedade do MovingObject do TGLNavigador deve ser configurada para GLCamera1, para que o usuário interaja com a câmera aproximando, afastando ou rotacionando a mesma. As propriedades GLNavigator e GLVerticalNavigator do objeto TGLUserInterface devem ser alteradas para GLNavigator1 e a propriedade MouseSpeed para 10. Também se deve adicionar a Unit GLKeyboard ao projeto para utilizar as funções do teclado que permitiram a interação. A **Listagem 1** mostra os códigos descritos anteriormente.

Vale à pena evidenciar que as propriedades dos objetos podem ser alteradas em tempo de execução, sendo assim, o desenvolvedor pode criar um efeito de animação simplesmente alterando tais propriedades da maneira desejada nos eventos desejados, lembrando que eventos de mouse e teclado também podem ser utilizados para criar uma interação com o usuário.

Para incrementar o ambiente gráfico da aplicação desenvolvida é interessante adicionar um efeito de luz por exemplo. Deve-se abrir o editor da cena clicando duas vezes no objeto GLScene1, selecionar Scene objects e clicar com o botão direito. No menu Add object basta selecionar a opção LightSource. O efeito de luz já está adicionado de uma maneira bem simples. É evidente que ele foi criado com suas propriedades definidas como padrão. Para alterar a direção da luz, por exemplo, basta selecionar o objeto GLLightSoruce1 presente no editor e ajustar a coordenada Y = 5 na propriedade Position para que a luz venha do topo. Para alterar a cor padrão de fundo da janela de visualização para a cor preta, basta selecionar o GLSceneViewer e alterar a sub-propriedade BackgroundColor para clBlack presente na propriedade Buffer.

Como foi observado até o momento para criar uma cena 3D com a GLScene bastaram alguns cliques, alguns ajustes de propriedades e os objetos já estavam prontos. A utilização de código foi mínima.

Quando utilizamos uma câmera, a propriedade NearPlaneBias deve ser alterada para 0,1, eviando que a câmera penetre no objeto quando for aproximada, cortando o mesmo e gerando um efeito indesejado.

Vamos alterar as cores do cubo. Abrindo o Scene Editor do objeto GLScene dando um clique duplo e selecionar o objeto GLCube1 adicionado anteriormente. Agora basta abrir o editor de materiais para adicionar a cor desejada ao Cubo através do botão adicional da propriedade Materials. A **Figura 1** mostra o resultado da aplicação no final de todos os passos descritos acima.

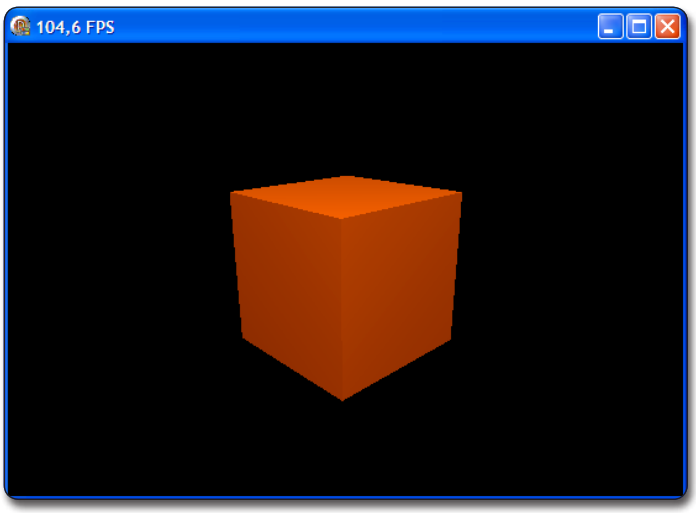


Figura 1. Cubo 3D animado na Aplicação Delphi (VCL)

Listagem 1. Adicionando uma animação 3D ao Cubo

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  GLUserInterface1.MouseLookActivate;
end;
procedure TForm1.GLCadencer1Progress(Sender: TObject; const deltaTime,
  newTime: Double);
var
  velocidade: Real;
begin
  GLCube1.TurnAngle := 90 * newTime;
  Form1.Caption := GLSceneViewer1.FramesPerSecondText;
  GLSceneViewer1.ResetPerformanceMonitor;
  velocidade := 10;
  if IsKeyDown(VK_UP) then
    GLNavigator1.MoveForward(velocidade*deltaTime);
  if IsKeyDown(VK_DOWN) then
    GLNavigator1.MoveForward(-velocidade*deltaTime);
  if IsKeyDown(VK_ESCAPE) then
    Close;
  GLUserInterface1.MouseUpdate;
  GLUserInterface1.MouseLook;
end;
Form1.Caption := GLSceneViewer1.FramesPerSecondText;
GLSceneViewer1.ResetPerformanceMonitor;
```

Conclusão

A OpenGL é uma biblioteca gráfica rica presente em diversas plataformas. Apesar de ter surgido á vários anos atrás, é uma biblioteca muito implementada na computação gráfica atual e utilizada nos mais novos dispositivos portáteis. Permite ao desenvolvedor da aplicação a utilização de efeitos gráficos simples e complexos, em 2D ou 3D, de alta qualidade e desempenho, porém envolve um certo conhecimento e familiarização com suas funções e propriedades. No entanto existem componentes que a implementam, aumentando assim a produtividade e facilitando a vida de quem necessite ou deseje trabalhar com 3D, como é o caso da GLScene. A GLScene é uma biblioteca destinada ao Delphi completa que abstrai as complexas funções da OpenGL e oferece uma interface mais amigável ao desenvolvedor, para que o mesmo possa atingir seus objetivos de uma maneira mais simples e rápida. Como visto, com apenas alguns componentes, criação de alguns objetos, ajustes de propriedades, interligações entre os objetos e pouco trecho de código é possível criar uma cena gráfica simples e permitir que o usuário interaja com ela, em outras palavras, com apenas alguns minutos e cliques do mouse é possível se obter uma cena 3D, diferente de se trabalhar diretamente com OpenGL onde o desenvolvedor teria que desprender mais tempo e conhecimento. Evidente que para aqueles que desejam desenvolver um jogo baseado em OpenGL com Delphi, a GLScene é altamente indicada, uma vez que é considerada uma engine, e como visto, permite a utilização de modelos de personagens e objetos, terrenos, luzes, efeitos, detecção de colisão, áudio, interação com usuário entre várias outras funções.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Download da Unit dglOpenGL
http://vgasoft.googlecode.com/svn/OpenGL/dglOpenGL.pas

Site do Projeto GLScene
http://glscene.sourceforge.net/wikka/HomePage

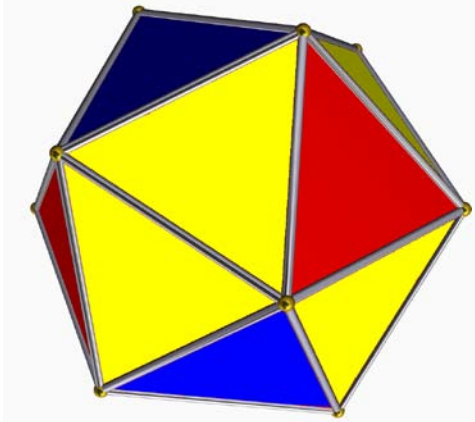
Download da GLScene para Delphi
http://sourceforge.net/projects/glscene/files/

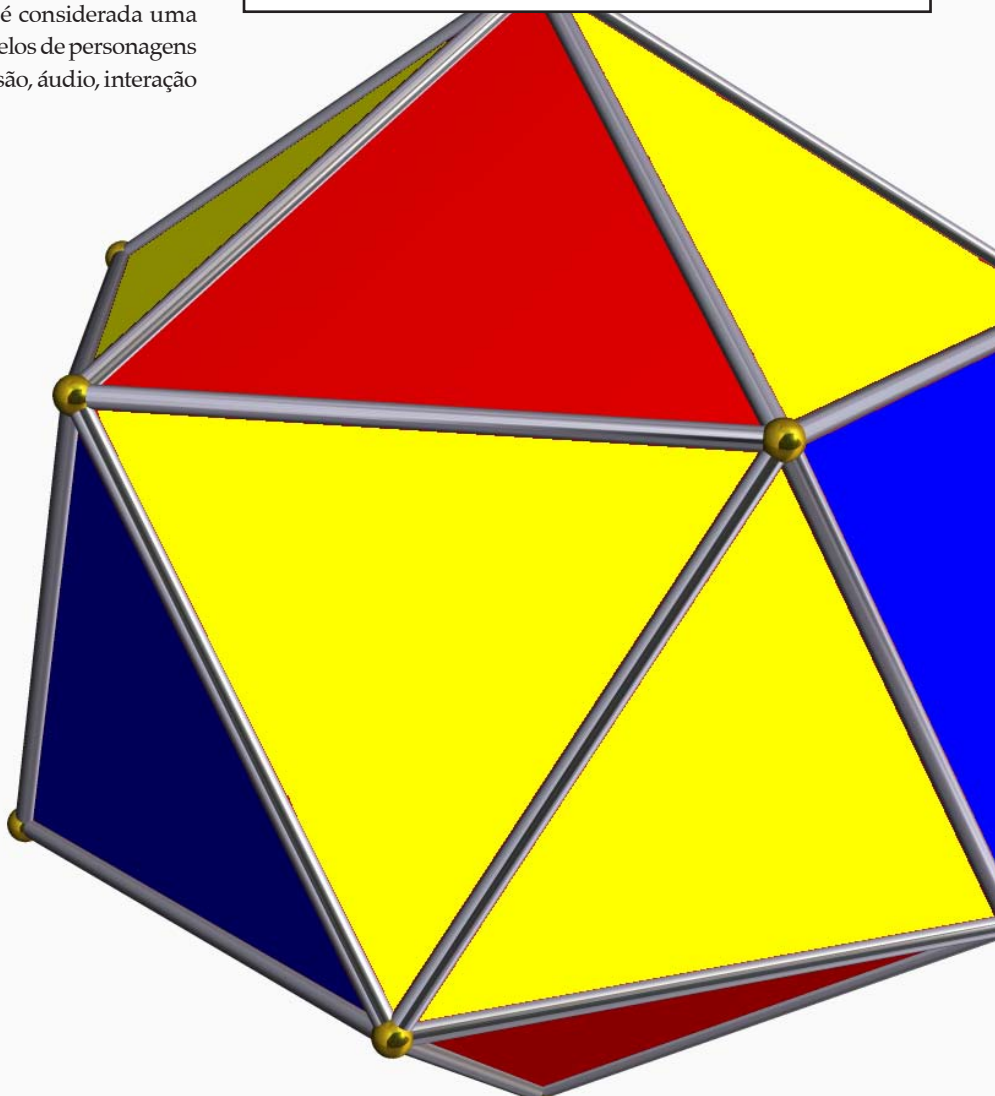
Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback





FORUM DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo o que acontece nas novas tecnologias do mercado de programação

No novo Forum da DevMedia você vai encontrar canais específicos de Delphi, ASP.NET, Java, Banco de Dados e Engenharia de Software; além de ter contato com profissionais qualificados da área para troca de informações, sugestões e MUUUITO MAIS...

ACESSE AGORA
www.devmedia.com.br/forum

DevMedia

O Poderoso Firebird – Entendendo e criando banco de

Resumo DevMan

De que trata o artigo:

O artigo mostrará a criação de um arquivo de banco de dados, como criar tabelas, chave primária, chave estrangeira (integridade referencial) e índices. Veremos também comandos de CRUD (insert, update, delete e select).

Em que situação o tema é útil:

Na administração básica de um banco de dados, pois no dia a dia geralmente há a necessidade de criar tabelas e integridades referenciais para que os dados possam ser consistentes. Será visto aos poucos, o quanto o Firebird é poderoso e pode ajudar nas soluções corporativas da sua empresa.

O Podereso Firebird – Parte 2:

Este artigo irá mostrar a administração básica do servidor Firebird, ou seja, iremos aprender como criar um banco de dados de forma correta para termos uma melhor administração e maior desempenho do mesmo. Além disso, iremos aprender a criar tabelas e seus campos, veremos os tipos de dados que o Firebird suporta, aprenderemos a criar a chave primária e chave estrangeira (integridade referencial) e como criar os índices para agilizar as consultas (recuperação de dados) no dia a dia. Veremos também, como popular um banco de dados ou tabela e como realizar alterações e apagar os dados.

Pode-se classificar os bancos de dados em diferentes categorias como a seguir:

- Bancos de dados arquivo-servidor e cliente-servidor;
- Bancos de dados relacionais;
- Bancos de dados OLTP e OLAP.

Bancos de Dados Arquivo-Servidor

Em um banco de dados arquivo-servidor os dados são armazenados e um arquivo e os usuários dos dados obtêm o que eles precisam diretamente do arquivo, ou seja, quando há a necessidade de incluir, alterar ou excluir alguma informação, o usuário através de algum aplicativo (cliente) abrirá o arquivo e fará as operações de CRUD diretamente. Isso para cada usuário que precisar fazer alguma coisa no arquivo. Um exemplo deste tipo de banco de dados é o Microsoft Access e o Microsoft Fox Pro.

Bancos de Dados Cliente-Servidor

Já em um banco de dados Cliente-Servidor,os dados continuam sendo armazenados em um arquivo, mas todos os acessos ao arquivo são controlados por um único programa, o servidor. Quando um aplicativo (cliente) precisar fazer uma consulta ou qualquer outra operação do CRUD, o aplicativo não fará diretamente no arquivo, mas mandará uma requisição à aplicação servidora e a mesma fará todas as operações necessárias. A vantagem de utilizar banco de dados Cliente-Servidor são várias, algumas delas são:

- Pelo fato dos dados serem acessados por um único programa, há menos possibilidades de que alterações ou travamentos acidentais alterem os dados;
- Eficiência em usar a banda disponível da rede para se comunicar;
- Aumento do desempenho;
- Controle de múltiplas transações.

Parte 2
dados

Fabiano Moura

Nota do DevMan

CRUD é um acrônimo de Create, Retrieve (Read), Update e Delete na língua inglesa. Se representar em SQL seria Insert, Select, Update e Delete. Veja essa representação:

C → Create => INSERT (inserir ou criar novas entradas);

R → Retrieve => SELECT (recuperar, consultar ou ler os dados);

U → Update => UPDATE (alterar os dados já salvos);

D → Delete => DELETE (apagar os dados já salvos).

Existem vários exemplos de banco de dados Cliente-Servidor, alguns deles são: Oracle, SQL Server, Interbase, Firebird, entre outros.

Bancos de Dados Relacionais

Um banco de dados relacional tem a capacidade de armazenar os dados em várias tabelas, enquanto também monitora como se relacionam. O banco de dados relacional possui a capacidade de guardar dados únicos através de campos (field) chaves, conhecidos como chaves primárias (primary key) e de relacionar tabelas através de campos estrangeiros, conhecidos como chave estrangeira. Através destas chaves há o controle da chamada integridade referencial. Os bancos relacionais também são chamados de SGBDR ou seja, Sistema Gerenciador de Banco de Dados Relacional e alguns exemplos são: DB2, Oracle, SQL Server, MySQL, Interbase, Firebird, entre outros.

Embora os bancos de dados relacionais sejam muito flexíveis, só existe uma única forma de garantir a integridade dos dados e desempenho satisfatório: um bom projeto de banco de dados. Um bom projeto de banco de dados deve seguir algumas regras, para garantir alguns requisitos de qualidade:

- Garantir que o que é pedido para o usuário, os dados contidos têm condições de oferecer;
- Garantir a integridade dos dados não deixando por exemplo apagar um registro caso o mesmo esteja vinculado com outro;
- Ser de fácil compreensão;
- Satisfazer de forma ideal os requisitos de velocidade;

Bancos de Dados OLTP e OLAP

Outro conceito bastante encontrado no mercado são os bancos de dados OLTP e OLAP. OLTP significa processamento de transações on-line. OLAP significa processamento analítico on-line.

Nota do DevMan

O DBA antes de criar o banco de dados e seus objetos procura desenvolver um projeto, utilizando diagramas, aonde são discriminados as tabelas, seus nomes e os campos que possui. Além disso as chaves primárias e as ligações com as outras tabelas, através de campos estrangeiros, ligados ao campos chaves. Além do diagrama também é criado um dicionário de dados aonde explicará o que cada tabela representa, o que cada campo significa e os tipos de dados que o campo receberá.

As distinções desses bancos não são tão claras como os bancos arquivo-servidor e cliente-servidor, pois na verdade, a grande maioria dos bancos de dados serão usados como produtos OLTP e OLAP.

OLTP é um padrão de uso envolvendo inserções, atualizações e remoções rápidas de dados. O que é comum em muitos aplicativos. Pense no seguinte exemplo: Você vai fazer uma viagem de turismo e vai até uma agência, comprar a passagem, bilhete do ônibus ou avião. A capacidade de fazer a consulta rápida para ver se ainda há vagas para o dia que você deseja e a capacidade de incluir estes dados de forma rápida é a função OLTP. Agora pense que você é o gerente desta agência e gostaria de saber o perfil das pessoas que estão viajando, mas de uma forma agrupada por dia e por sexo. Essa capacidade de busca é a marca conhecida dos aplicativos OLAP. Os aplicativos OLAP tem por finalidade trazer dados resumidos, para tomadas de decisões rápidas por parte da empresa.

Um servidor de banco de dados pode ser tanto OLTP quanto OLAP, porém, é muito difícil um servidor de banco de dados ser as duas coisas ao mesmo tempo, pois as estruturas de dados que são apropriados para fazer uma inserção, atualização ou remoção de dados rápida e segura, não são estruturas rápidas para consulta. Para resolver esse problema há várias possibilidades, desde utilizar objetos do banco de dados ou até mesmo criar um banco de dados a parte.

Criando um banco de dados

Muitos desenvolvedores iniciantes ficam presos aos IDEs visuais para criação e administração de bancos dados. O Firebird disponibiliza utilitários de linha de comando para esse fim, dessa forma, na ausência de algum gerenciador visual podemos interagir com o um banco de dados de forma textual. Ao acessar a pasta Bin do Firebird (via prompt de comando) é possível utilizar o comando isql para iniciar um ambiente que executará comandos SQL, basta digitar isql e pressionar ENTER. Uma vez em execução observe que o prompt de comando é alterado para SQL>, isso indica que não estamos mais em um prompt comum, mas que estamos

utilizando o utilitário Interactive SQL. Para se criar um banco de dados nesse ambiente utiliza-se o seguinte comando:

```
CREATE DATABASE "C:\DEVEMEDIA.GDB" USER "SYSDBA" PASSWORD "MASTERKEY";
```

O comando CREATE DATABASE recebe como parâmetro o local onde deve ser criado o arquivo do banco de dados e o login e senha do usuário com permissão de administrador. SYSDBA/MASTERKEY são os valores padrão, talvez você utilize credenciais diferentes, assim, substitua esse usuário pelo o que você utiliza. Todo comando digitado no ISQL deve encerrar com um ponto e vírgula, indicando o fim do comando.

Tipos de dados

O Firebird, como a grande maioria dos bancos de dados relacionais, necessita de tipos de dados para poder definir e tipar os seus campos. Podemos citar dois exemplos: No primeiro exemplo em um determinado campo de uma tabela de produtos será necessário guardar a quantidade em estoque deste produto. Como quantidade em estoque são valores matemáticos, não é necessário então que neste devido campo haja informações de caracteres texto, mas somente informações de caracteres numéricos. Em outro exemplo pode-se considerar um campo que irá armazenar nomes de pessoas. Nomes de pessoas devem aceitar caracteres de texto.

Desta forma os tipos de dados definem qual é o tipo de informação que será armazenada em um determinado campo de uma tabela do banco de dados. O Firebird disponibiliza diversos tipos de dados que podem ser utilizados nos mais variados tipos de campos presentes em uma determinada aplicação. Na **Tabela 1** estão relacionados os tipos de dados mais utilizados no Firebird.

Criando Tabelas

As tabelas em um banco de dados são utilizadas para agrupar valores comuns, ou seja, se é necessário que o sistema controle as informações de determinados clientes, necessita-se da criação de um lugar para armazená-los. Para criar uma tabela no banco de dados Firebird é necessário utilizar comandos em SQL ou utilizar uma ferramenta que gera o script desses comandos para o banco de dados.

Todos os comandos que mexem com a estrutura de um banco de dados são chamados de D.D.L. (Data Definition Language ou Linguagem de Definição de dados). Este tipo de linguagem altera a estrutura do banco de dados, permitindo criação de tabelas, índices, procedimentos armazenas e muito mais. No código da **Listagem 1** está o exemplo da criação de uma tabela para armazenar o nome de uma pessoa, o telefone, o celular e o email das pessoas.

Podemos comentar alguns pontos relativos a esta criação como a instrução not null que informa que o campo código não poderá ser nulo, ou seja, sempre deverá conter um determinado valor. Também é observado o primary key(codigo) que cria a chave primária desta tabela. Após a criação desta tabela ela já está pronta para receber valores, ou seja, já é possível preencher as informações desejadas em cada um dos campos.

Nota do DevMan

Outra forma de criar um banco de dados é utilizar o IBExpert que é bem difundido entre os usuários do Firebird. O IBExpert é um gerenciador completo para banco de dados Firebird que permite realizar de forma visual criação de objetos do banco de dados (tabelas, procedures), realizar consultas SQL, executar diagnósticos, depurar procedimentos armazenados, realizar backups e muito mais. É uma ferramenta comercial, porém existe uma versão gratuita com menos recursos, mas totalmente útil. Após seu devido download (veja seção Links) e instalação que segue o formato de instalação Windows (Next, Next, Finish), acessamos seu menu Database>Create Database. Na janela que se abre pode-se informar os seguinte parâmetros:

- Server: É informado se será utilizada conexão remota ou conexão local durante o acesso ao sistema gerenciador do banco de dados. Caso seja conexão local são suprimidas as informações server name e protocol, visto que é utilizado um protocolo especial para conexão local. No caso de ser remoto, é necessário informar o endereço ou nome do host.
- Server Name: Nome ou endereço do host que deverá estar rodando umas das versões do Firebird.
- Protocol: Protocolo de comunicação, geralmente é utilizado o protocolo tcp/ip
- Database: Caminho aonde será criada a base de dados.
- Client Libary File: Biblioteca do cliente Firebird, é a gds32.dll ou fbclient.dll.
- Username: Nome do usuário, geralmente é utilizado o usuário padrão que é o sysdba.
- Password: Senha do usuário Informado.
- SQL Dialect: Dialeto do Banco de dados, atualmente o dialeto utilizado é o 3.
- Page Size: Tamanho da pagina de dados.
- Charset: O charset é como se fosse a língua do banco de dados, aqui no Brasil é utilizado WIN1252 ou ISO8859_1. Também pode ser suprimida esta informação.
- Collation (FB 2.5): É especificada qual a codificação que será usada no banco de dados.

Listagem 1. Criando uma tabela de pessoas

```
Create table pessoa (
  Codigo integer not null,
  Nome varchar(80),
  Telefone varchar(20),
  Celular varchar(20),
  Email varchar(20),
  Primary key(codigo)
);
```

Campo calculado

Campos calculados são campos que o próprio Firebird se encarrega de gerar seu conteúdo baseando-se em expressões que foram definidas na criação desses campos, evitando que o desenvolvedor fique se preocupando com essas informações. Para entender o que é um campo calculado considere que em uma determinada tabela do banco de dados separam-se informações de dinheiro e cheque em campos diferentes, no entanto, em um determinado relatório se faz necessário adquirir os 2 valores somados. Considerando esta situação é possível definir já na criação da tabela, ou até mesmo após ela, um campo calculado fazendo a soma deste dois valores. No exemplo da **Listagem 2** é apresentado o valor total do estoque de um determinado item, multiplicando a quantidade de estoque pelo valor de venda.

Tipo	Descrição	Valores Aceitos	Sintaxe
Blob	Neste tipo de dado podem ser gravados diversos tipos de informações, desde arquivos binários até informações textuais	Aceita qualquer tipo de informação, imagens, binários, textos.	<nome_campo> blob sub_type <Tipo do Blob>
Char	Este tipo de dados é utilizado para gravar informações alfanuméricas, é necessário definir um tamanho máximo de caracteres que ele poderá gravar. Uma vez definido o tamanho e não utilizando na sua totalidade, o espaço é utilizado.	Números, letras, caracteres especiais,	<nome_campo> char(N) *N = tamanho definido
VarChar	Este tipo de dado tem as mesmas características do tipo Char, no entanto, se um tamanho for definido e não for utilizado ele não ocupará todo o espaço reservado para o campo.	Números, letras, caracteres especiais,	<nome_campo> varchar(N) *N = tamanho definido
Date	O tipo de dado Date é utilizado para gravar informações de data.	Datas válidas	<nome_campo> date
Time	O tipo de dado Time é utilizado para gravar informações de horários	Hora válidas	<nome_campo> time
TimeStamp	O tipo de dado TimeStamp é utilizado para gravar informação de data e hora, juntas.	Data e Hora Válidas	<nome_campo> timestamp
Decimal	O tipo de dado Decimal é utilizado para gravar informações numéricas com um tamanho fixo e um número de casas após a vírgula. Vale salientar que o tamanho do campo for definido como 15 e a precisão em 2 então terá 13 espaços para números inteiros e 2 para números após a virgula.	Valores	<nome_campo> decimal(N,P) N = Tamanho do campo P = Precisão a ser utilizada
Numeric	Internamente este tipo de dado tem as mesmas características do tipo DECIMAL	Valores	<nome_campo> numeric(N,P) N = Tamanho do Campo P = Precisão a ser utilizada
SmallInt	Tipo de dado que suporta números inteiros entre -32.768 até 32.767	Valores inteiros	<nome_campo> smallint
Integer	Tipo de dados que suporta números em uma faixa de valores bem maior que o smallint, de -2.147.483.648 até 2.147.483.647.	Valores inteiros	<nome_campo> integer
Double Precision	Tipo de dados que suporta valores de ponto flutuante sem especificar a precisão.	Valores Reais	<nome_campo> double precision

Tabela 1. Alguns tipos de dados suportados pelo Firebird no dialeto 3

Listagem 2. Criação de uma tabela com um campo calculado

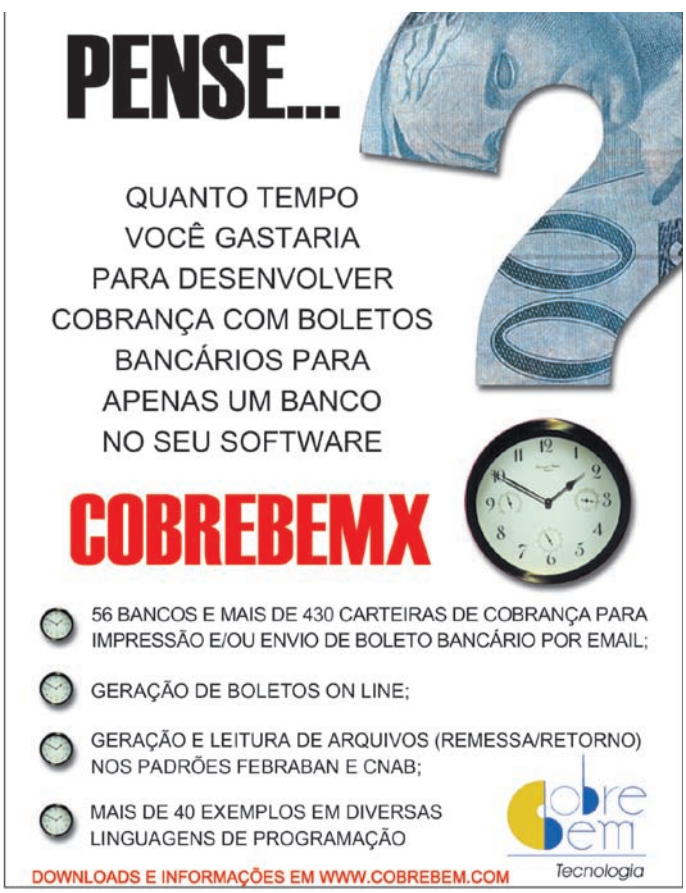
```
create table produtos (
 codigo integer not null,
 descricao varchar(100),
 qtde numeric(15,3),
 valorvenda decimal(15,2),
 total computed by(qtde*valorvenda),
 primary key(codigo)
);
```

O campo total que é computado pelo valor de qtde*valorvenda, ou seja, quando for inserido um registro, a multiplicação desses dois valores será preenchida automaticamente.

Vale salientar que quando houver uma alteração o servidor de banco de dados se encarregará de fazer as devidas atualizações para que os desenvolvedores não fiquem se preocupando com esses dados.

Chave primária

Chave primária de uma tabela nada mais é do que um ou mais campos de dados que formam um índice que não pode ter seus valores repetidos e também não podem ser nulos. É ela quem define a unicidade de identificação de um determinado registro em uma tabela do banco de dados. Na realidade todos os outros campos da tabela podem ter o mesmo valor, mas do ponto de vista real eles são registros diferentes pois suas chaves primárias não são as mesmas.



Para criar uma chave primária basta utilizar a instrução primary key quando estiver efetuando a criação de uma determinada tabela. Sempre que um campo for escolhido para ser uma chave primária é necessário que este campo também contenha a instrução not null.

A chave primária também é muito importante já que é ela que é relacionada com outras tabelas, e sendo assim, é possível identificar o registro em questão em outras tabelas também. A chave primária também pode ser formada por mais de um campo, bastando inserir dentro do parênteses esses campos separados por vírgula

Chave estrangeira

Chave estrangeira é um determinado campo em uma tabela que faz a relação com outra tabela para manter os dados coesos e relacionados. Para conseguir entender melhor o conceito, no exemplo a seguir (**Listagem 3**) são apresentadas duas tabelas de dados que fazem relação entre si, estas tabelas são as tabelas cliente e telefones do cliente. A ideia aqui é fazer uma estrutura aonde um único cliente poderá ter mais de um telefone.

Listagem 3. Criando as tabelas e a relação entre elas

```
01 create table cliente (
02   codigo integer not null,
03   nome varchar(60),
04   primary key(codigo)
05 );
06
07 create table telefone_cliente (
08  id integer not null,
09  codcliente integer,
10  telefone varchar(20),
11  primary key(id),
12  foreign key(codcliente) references cliente(codigo)
13 );
```

Observa-se que na linha 12 existe uma instrução que até agora não foi vista, que é o comando foreign key(codcliente) references cliente(código). Este comando tem a responsabilidade de indicar que a tabela telefone_cliente vai ter um campo estrangeiro que é codcliente e este campo irá fazer referência para o campo código da tabela cliente. Sabe-se pela instrução de criação da tabela de cliente que o campo codigo é chave primária, já que existe a instrução primary key.

A maior vantagem deste tipo de relação é que, se alguém quiser fazer uma exclusão de um determinado cliente e este tiver telefones cadastrados, o banco de dados emitirá uma exceção impossibilitando a operação, já que existem registros em outras tabelas que dependem da existência deste registro. Esta operação chama-se integridade referencial e é uma das grandes vantagens de sistemas gerenciadores de bancos de dados relacionais, visto que a consistência dos dados é garantida através disto. No caso para poder apagar este cliente deve-se necessariamente excluir todos os outros registros que dependem dele, e após isso, excluir o cliente em si. Esse é o comportamento padrão, contudo ao criar uma chave estrangeira podem-se definir operações de cascata. Por exemplo, ao excluir um cliente seus telefones seriam automaticamente excluídos. Assim, a linha 12 seria alterada para:

```
foreign key (codcliente) references cliente(codigo) on delete cascade
```

Digamos a chave primária de um cliente possa ser alterada. Na criação padrão isso não seria possível por conta a integridade referencial, contudo, é possível também determinar na chave estrangeira que, caso sua tabela de referência seja alterada, seu campo de ligação também seja atualizado. Assim, para incluir essa opção a linha 12 seria alterar para:

```
foreign key (codcliente) references cliente(codigo) on update cascade
```

Vale mencionar que você pode registrar os dois comportamentos ao mesmo tempos:

```
foreign key (codcliente) references cliente(codigo) on delete cascade on update cascade
```

Criação de índices

Os índices são estruturas que são utilizadas pelo motor do Firebird para acelerar a localização de registros em um banco de dados. Eles podem ser compostos ou não, podem ser ordenados de forma ascendente ou descendente. Tomando por base a tabela cliente podemos criar um índice para a coluna nome da seguinte forma:

```
CREATE INDEX CLIENTE_IDX_NOME ON CLIENTE (NOME)
```

Para o comando Create Index passamos o nome índice a ser criado e logo depois do comando On informamos o nome da tabela e qual campo que será indexado. Esses índices também podem ser removidos, assim, para remover o índice na tabela cliente usaríamos o seguinte comando:

```
DROP INDEX CLIENTE_IDX_NOME
```

Operações de CRUD

Para inserir informações em uma tabela utilizamos o comando INSERT, ele possui a seguinte sintaxe:

```
Insert into [tabela] values [(valores)]
```

Assim, para inserir um cliente podemos executar o seguinte comando:

```
Insert into Cliente values (1,'JOSE');
```

O comando ainda possui alguns detalhes. Como que o Firebird sabe que o número 1 é para ser jogado na coluna código e o texto JOSE na coluna nome? O comportamento padrão do SQL é que se você não especificar nada é assumido que os valores (values) estão na mesma ordem das colunas. Agora, se você precisa informar

em outra ordem ou não irá informar todos os valores, a sintaxe é um pouco diferente:

```
Insert into Cliente(Nome, Codigo) values ('JOSE',1);
```

Como se pode notar nessa situação foi necessário informar os campos que receberiam valor. Para atualizar um registro é o utilizado o comando UPDATE que possui a sintaxe a seguir:

```
Update [tabela]
Set [campo1] = [valor1] [, [campo2 = valor2], [lista de campos e valores]
Where [condição]
```

É necessário informar qual tabela que sofrerá a atualização e quais campos e valores serão alterados. Também é possível especificar uma condição que indique quais registros devem ser alterados. Aplicando isso à nossa tabela Cliente digamos que queremos mudar o nome do cliente 1 para 'ALFREDO', a seguinte instrução deve ser implementada:

```
Update Cliente
Set nome = 'ALFREDO'
Where Codigo = 1;
```

Dessa forma o Firebird irá abrir a tabela Cliente, localizar o registro que possui o código 1 e atualizar o campo nome.

Aqui já começamos a ver algo sobre pesquisa. De uma forma simples registros de uma tabela são recuperados através de um comando SELECT:

```
Select [campo] from [tabela] where [condição] [order by [campos]]
```

Essa é uma sintaxe bem simples de um comando SELECT. Informamos a lista de campos que desejamos trazer, o nome da tabela que contém esses campos e uma possível condição que filtre os dados. Esses dados ainda podem ser organizados por campos da tabela. Assim, podemos realizar uma pesquisa de clientes, retornando todos que possuem o nome iniciando com "RICARDO":

```
Select * from Cliente where nome like 'RICARDO%' order by nome;
```

Nessa sentence não relacionamos os campos, assim utilizamos o caractere * para indicar que todos os campos da tabela devem ser trazidos. Na cláusula where foi utilizado um comando de busca textual chamado like. Com ele é possível fazer uso de máscaras para indicar o texto que se deseja buscar. A posição dessa máscara indica se a busca será algo como "iniciando com", "contendo" e "finalizando com". A primeira situação foi expressa já, as outras podem ser vistas a seguir:

```
Select * from Cliente where nome like '%RICARDO%' order by nome;
Select * from Cliente where nome like '%RICARDO' order by nome;
```

O resultado foi ordenado pela coluna nome, essa ordenação também pode ser descendente ou ascendente (padrão), basta fazer uso do comando DESC ou ASC após especificar o campo no order by.

Para encerrar temos a exclusão. Como é de se imaginar o comando SQL que realiza essa operação é o DELETE:

```
Delete from [tabela] where [condição];
```

Este comando é muito simples porém se não aplicado de forma correta pode fazer um grande estrago em sua base de dados, isso porque uma vez realizado o comit dos comandos, eles não podem ser mais desfeitos. Assim é importante assegurar que a cláusula where de um comando delete esteja correta. Se desejarmos excluir do banco de dados o cliente de código 1 a seguinte sintaxe seve ser empregada:

```
Delete from cliente where código = 1;
```

Conclusão

Este artigo não tinha a pretensão de ser um manual de todos os utilitários e comandos SQL, mas apenas a intenção de posicionar novos desenvolvedores com os recursos disponíveis por este banco de dados. Muito mais aqui merece ser assunto de estudo. Aplicativos como GBak, Gfix e os próprios comandos SQL exibidos. Agora é com você leitor. Até uma próxima.

Fabiano Moura
fabiano@fabianomoura.com.br
http://fabianomoura.com.br
Analista de Sistemas no Hospital Estadual de Américo Brasiliense – HEAB. Desenvolve sistemas hospitalares utilizando conceitos ERP e as tecnologias Delphi e Oracle. Graduado na FATEC Taquaritinga – SP e fazendo MBA em Gestão da Qualidade de Software na UNIARA, Araraquara – SP. Embaixador do MindTheBird, aonde possui a missão de evangelizar e divulgar o Firebird a todas as pessoas.

Firebird
www.firebirdsql.org

IBExpert
www.ibexpert.com

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

FireMonkey

Introdução ao desenvolvimento com ba

Resumo DevMan

De que trata o artigo:

Conforme apresentado recentemente aqui nesta revista, a nova versão do Delphi, a XE2, foi lançada e veio repleta de novidades. Uma que chamou a atenção da comunidade antes mesmo do lançamento oficial foi o tão badalado FireMonkey. Este que é o mais novo framework da Embarcadero voltado para o desenvolvimento de aplicações de negócios com visual mais rico e de caráter multi-plataforma, rodando nativamente em Windows, Mac (OS X) e iOS. Dessa forma, este artigo visa fazer uma explanação de como se dá a construção de aplicações com o FireMonkey, e sua proximidade e diferenças com relação às aplicações VCL tradicionais.

Em que situação o tema é útil:

Se você é um desenvolvedor Delphi que almeja melhorar suas aplicações, seja no aspecto visual, através de interfaces de usuário mais ricas, ou ainda torná-las aptas a rodarem em outros ambientes além do Windows, como Mac (OS X) e iOS, a alternativa ideal se resume em apenas um nome: FireMonkey. Esta nova plataforma de desenvolvimento de aplicações da Embarcadero traz consigo uma nova gama de componentes, disponibilizados no IDE através da tradicional Tool Palette, proporcionando ao desenvolvedor a mesma experiência RAD que consagrou o Delphi.

Dessa forma, além do reaproveitamento do conhecimento, o profissional consegue facilmente construir aplicações que atendam a novos nichos de mercado.

FireMonkey:

O FireMonkey é considerado pela Embarcadero figura principal da nova geração de plataformas para desenvolvimento de aplicações. Aplicações estas que apresentam como principais características o seu caráter multi-plataforma, ou seja, podem ser executadas em ambientes distintos além de possuir um visual mais arrojado e rico. Este tipo de visual proporciona um envolvimento maior com o usuário, potencializando seu grau de experiência. Tecnicamente falando, os recursos gráficos do FireMonkey fazem uso da GPU do hardware, que é a unidade de processamento gráfico da máquina, possibilitando assim o trabalho com gráficos HD, 3D, efeitos de imagem e animações.

Outro recurso disponível é sua conectividade com banco de dados, utilizada principalmente em aplicações de negócios. Para estes casos entra em cena o LiveBindings que é uma nova forma de ligação de dados apresentada também nesta versão XE2 do Delphi. Ao ser construída, uma aplicação FireMonkey pode ser direcionada nativamente para as plataformas Windows (32 e 64 bits), Mac (OS X) e iOS.

nco de dados nesta nova plataforma

Fabrício Hissao Kawata

Conforme dito, o FireMonkey é apresentado como sendo de natureza multi-plataforma, ou seja, presume-se que ele rode em outras plataformas além da já tradicional Windows 32 bits (Win32). Assim sendo, uma aplicação construída com FireMonkey poderá rodar também em Windows 64 bits, Mac OS X e iOS. Outro detalhe diz respeito à premissa de que tais aplicações possuem um visual mais rico que o habitual, que se torna um atrativo a mais ao usuário final. Devido a esta característica aplicações FireMonkey necessitam e fazem uso da GPU (Unidade de Processamento Gráfico) do hardware.

Todavia, independente das "novas características" que uma aplicação possa ter, ela sempre estará direcionada a um contexto de negócio. Exemplo disso é a construção de uma aplicação somente para fins de estudo, ou para elaboração de um aplicativo utilitário e ainda nos casos mais corriqueiros, aplicações comerciais. Neste ponto todas se parecem em um ponto fundamental que é o seu trabalho com dados, ou seja, as informações pertinentes do negócio que a aplicação está inserida. Em situações como esta, é comum a utilização de banco de dados para o armazenamento dos dados e consequentemente o vínculo deste com o aplicativo. No Delphi, este tipo de atividade é feito, tradicionalmente, com as aplicações do tipo VCL Forms Application. Contudo, com a chegada do FireMonkey, ganha-se mais uma alternativa de desenvolvimento nativo e é justamente neste ponto que este artigo pretende dar o start necessário ao desenvolvedor que queira dar os primeiros passos nesta nova plataforma de desenvolvimento apresentada pela Embarcadero.

Trabalhando com banco de dados em aplicações FireMonkey

Para se ter uma noção clara do que esperar do FireMonkey para a construção de aplicações com conectividade a banco de dados, nada melhor do que fazer um comparativo com a forma que já estamos acostumados a trabalhar, ou seja, através do desenvolvimento de aplicações do tipo VCL Forms Application.

No Delphi, ao se construir aplicações deste tipo e que fazem acesso a banco de dados são utilizadas tecnologias responsáveis por estabelecer a conexão e prover toda a interatividade necessária entre aplicação e banco. Eles fazem o intermédio entre as duas partes e são representados pelos componentes de acesso a dados, providos tanto nativamente pelo IDE quanto por terceiros (comerciais ou não). São exemplos nativos os componentes dos grupos dbExpress, DataSnap, InterBase e dbGo, cada qual com sua especialidade e propósito. Por exemplo, se a aplicação fará acesso a um banco Firebird, a opção mais sensata é fazer uso dos componentes DBX, do grupo dbExpress, que possui um driver específico para o SGBD. Na mesma medida, se o banco em questão for Interbase, da própria Embarcadero, os componentes IBX do grupo Interbase são os mais otimizados para o processo. Para SQL Server, os componentes ADO, do grupo dbGo são os mais tradicionais. Finalmente, caso a aplicação faça parte de uma estrutura multicamadas e, portanto, deverá acessar um servidor de aplicação, o uso de DataSnap é imprescindível.

Tomando como base a utilização do Firebird como SGBD e seu banco de dados de exemplo que acompanha sua instalação, denominado Employee, a definição inicial desta conexão numa aplicação VCL se dá no Data Explorer do próprio IDE. Este processo básico define o uso da tecnologia dbExpress. Quando usamos este tipo de abordagem, o famoso trio de componentes (SQLDataSet ou SQLQuery + DataSetProvider + ClientDataSet) se faz presente na maioria dos casos, juntamente de um componente de conexão central (SQLConnection). Meramente como demonstração a seguir temos na **Listagem 1** o código de um objeto de conexão SQLConnection, mapeado para o banco de dados mencionado.

Listagem 1. SQLConnection

```
object Employee: TSQLConnection
  ConnectionName = 'Employee'
  DriverName = 'FIREBIRD'
  GetDriverFunc = 'getSQLDriverINTERBASE'
  LibraryName = 'dbxfb.dll'
  LoginPrompt = False
  Params.Strings = (
   'drivername=FIREBIRD'
   'librarynameosx=libsqlfb.dylib'
   'vendorlibwin64=fbclient.dll'
   'vendorlibosx=/Library/Frameworks/Firebird.framework/Firebird'
   'blobsize=-1'
   'commitretain=False'
    'Database=localhost:C:\Program Files\Firebird\Firebird_2_5\exampl' +
    'es\empbuild\EMPLOYEE1.FDB'
   'localecode=0000'
   'Password=masterkey'
   'rolename=RoleName'
   'sqldialect=3'
   'isolationlevel=ReadCommitted'
   'user_name=sysdba'
   'waitonlocks=True'
   'trim char=False'
   'ServerCharSet=WIN1252')
  VendorLib = 'fbclient.dll'
end
```

Após isso, o processo natural se encaminha pela definição da tabela do banco a ser manipulada pela aplicação, através de um componente TSQLDataSet. Hipoteticamente ao optar pela tabela denominada Country, teríamos um objeto definido da seguinte forma:

```
object COUNTRY: TSQLDataSet
  SchemaName = 'sysdba'
  CommandText = 'select * from COUNTRY'
  DbxCommandType = 'Dbx.SQL'
  Params = <>
  SQLConnection = Employee
end
```

Finalizando, por motivos de boas práticas e além de poder trabalhar com os dados em memória, o uso de um TDataSet Provider e de um TClientDataSet acabariam por definir a estrutura inicial da aplicação, em termos de conectividade. Já para a manipulação em tela das informações obtidas da tabela do banco, em aplicações VCL temos a presença dos controles data-aware que têm a capacidade de ler e escrever valores para um determinado campo (Field) informado. Controles estes que podem ser encontrados nativamente no grupo Data Controls (**Figura 1**).

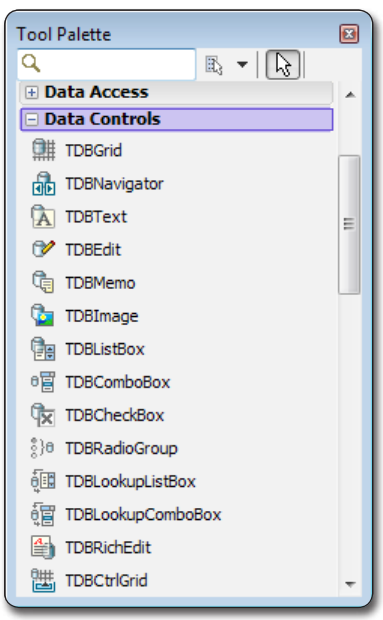


Figura 1. VCL - Data Controls

Tomando como base um simples TDBEdit, a definição do campo que se deseja manipular ocorre da maneira explícita através de sua propriedade DataField. A seguir temos o código do objeto que aponta o controle para um campo denominado CURRENCY, da tabela.

```
object DBEdit1: TDBEdit
  DataField = 'CURRENCY'
  DataSource = DataSource1
end
```

Fazendo um comparativo desta forma tradicional com as novas aplicações FireMonkey, a parte de conexão com banco de dados permanece inalterada, ou seja, é possível utilizar os mesmos grupos de componentes, sendo dbExpress e DataSnap as tecnologias recomendadas pela Embarcadero. Com isso, tem-se um reaproveitamento de conhecimento inicial. No entanto a grande diferença em relação ao desenvolvimento VCL diz respeito à forma que se dá a manipulação dos dados de uma tabela em tela. Isto porque o FireMonkey não possui controles data-aware, como os do grupo DataControls da VCL onde é possível simplesmente apontá-lo a um DataSet através de um TDataSource, e então selecionar determinado Field. Tomando como base o que foi mostrado até aqui, no FireMonkey ao invés de TDBEdit, temos um simples TEdit. Neste ponto a grande questão que inevitavelmente surge deve ser: é realmente possível construir uma aplicação FireMonkey com acesso a banco de dados? Se sim, de que forma isso é possível? A resposta para isso é LiveBindings.

Este novo recurso, como será visto adiante, faz com que os controles FireMonkey tornem-se data-aware e passem a se comportar conforme ao que já estamos acostumados em aplicações VCL. Isto acontece porque o FireMonkey utiliza-se de outros componentes LiveBinding para fazer o movimento de dados entre os componentes de acesso a dados e seus próprios controles. Dessa forma, o código do objeto TDBEdit que tínhamos na VCL, em FMX seria substituído por um TEdit simples, sem indicação de Field:

```
object Edit1: TEdit
  Position.Point = '(24,40)'
  Width = 100.000000000000000000
  Height = 22.000000000000000000
  TabOrder = 0
  ReadOnly = False
  Password = False
  Text = ''
end
```

Tal indicação se dará em um objeto de binding, conforme mostrado a seguir:

```
object DBLinkEdit1Currency1: TBindDBEditLink
    Category = 'DB Links'
    DataSource = BindScopeDB1
    FieldName = 'Currency'
    EditControl = Edit1
end
```

Neste caso, o tipo do objeto em questão é um TBindDBEditLink, que é o tipo utilizado para se conectar um controle FireMonkey a um Field de um componente de dados e que será visto em detalhes na parte prática do artigo. Ainda sobre o código apresentado, o detalhe fica por conta das propriedades EditControl e FieldName. A primeira aponta para o controle envolvido no binding enquanto que a segunda aponta para o nome do Field que se deseja vincular.

LiveBindings

Assim como o FireMonkey, o LiveBindings é uma das novidades da nova versão do Delphi, a XE2. Em suma, ele é o mais novo recurso para ligação de dados em aplicações Delphi, sejam elas VCL ou FireMonkey, e que se baseia em expressões para interligar objetos através de suas propriedades. Para quem utiliza ou já utilizou as tecnologias WPF e Silverlight, através do Delphi Prism, o recurso de binding destas é muito similar.

As expressões utilizadas pelo LiveBindings são do tipo relacional, conhecidas como Binding Expressions. Estas podem ser do tipo unidirecional ou bidirecional, que é a característica que define o fluxo dos dados na ligação. Normalmente, esta ligação se dá entre objetos distintos através de uma ou mais de suas propriedades, num processo que à grosso modo estabelece uma relação entre objetos de origem e destino, onde o primeiro é quem provê os dados ao segundo. No LiveBindings estes objetos são tecnicamente nomeados, respectivamente, como source objects (objetos de origem) e control objects (objetos de controle). Nesta mesma linha há ainda a possibilidade de se ter um objeto ligado a si mesmo, ou seja, o próprio objeto se coloca no papel de origem e destino.

Um exemplo bem corriqueiro deste tipo de funcionalidade é poder refletir instantaneamente o texto que é digitado em uma caixa de entrada (um TEdit ou um TMemo, por exemplo) em outro controle de exibição, como um TLabel. Na **Figura 2** temos o esboço de uma interface visual de uma aplicação VCL tradicional com esse propósito.

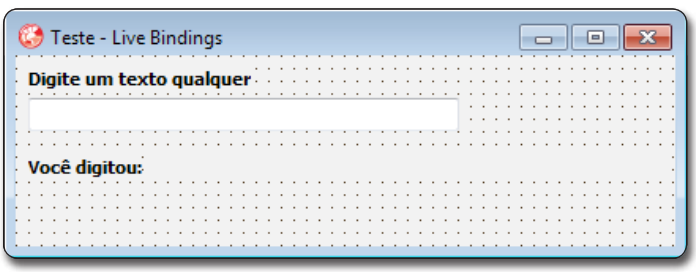


Figura 2. Interface visual – Aplicação VCL Forms

Na prática, o Delphi provê todo este processo de forma visual através do trabalho com propriedades chaves presentes nos componentes visuais de aplicações VCL e FireMonkey, via Object Inspector. Além destes o LiveBindings conta ainda com componentes específicos, disponíveis no novo grupo homônimo presente na Tool Palette (**Figura 3**). Obviamente é possível também definir toda a mecânica de binding programaticamente, ou seja, via código, para casos específicos. Por não ser tão intuitiva quanto à definição visual, a própria Embarcadero a recomenda somente para aplicações do tipo Console.

Para se ter uma noção clara do trabalho com LiveBindings e assim obter uma amostragem prévia do que acontece na prática numa futura aplicação FireMonkey, será feita uma explicação evolutiva, começando pela maneira mais tradicional de apresentá-los, que é através de uma aplicação do tipo VCL Forms Application. Desta forma, partindo da interface apresentada anteriormente serão mencionados rapidamente os passos básicos envolvidos para a definição da funcionalidade. Neste ponto, de acordo com a premissa da aplicação o objetivo a ser atingido é fazer com que a aplicação replique instantaneamente o texto digitado no controle TEdit (edtTexto) no controle TLabel (lblTexto). Tenha em mente que o controle TEdit será o objeto de origem (source object), enquanto que TLabel será o objeto de controle (control object).

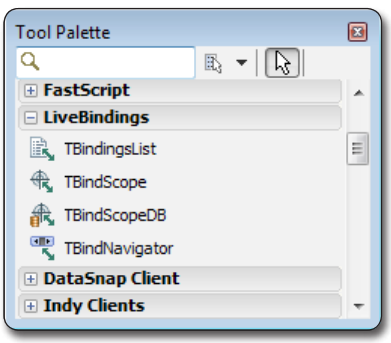


Figura 3. Novo grupo de componentes - LiveBindings

O passo inicial diz respeito à necessidade da presença de um componente TBindingList que serve para centralizarmos e gerenciarmos a lista de bindings criados, assim como acontece com o TActionList, que centraliza e gerencia ações presentes numa aplicação. Dito isto, é nele que se deve criar o novo objeto binding do tipo TBindExpression. Esta primeira etapa também pode ser iniciada a partir do objeto de controle através do menu de contexto de sua propriedade LiveBindings, conforme é mostrado na **Figura 4**. Fazendo isso, o próprio IDE se encarrega de mostrar uma caixa de diálogo com a lista dos possíveis tipos de objetos LiveBindings que podem ser criados (**Figura 5**), bem como em seguida adiciona automaticamente um componente TBindingList já com o novo Binding criado.

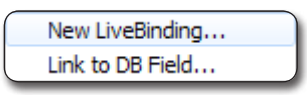


Figura 4. Menu de contexto

No TBindingList podemos visualizar o TBindingExpression criado (**Figura 6**). Dentre suas propriedades algumas merecem destaque, reforçando a teoria sobre LiveBindings apresentada inicialmente. SourceComponent e ControlComponent referem-se, obviamente, aos objetos de origem (source) e destino (control) envolvidos, neste caso, edtTexto e lblTexto respectivamente. Uma vez definidos os objetos faz-se necessária também a indicação da expressão a ser utilizada por cada um. Dessa forma, a propriedade ControlExpression que é a expressão relacionada a ControlComponent, recebe como valor "Caption", pois esta é a propriedade do componente TLabel (lblTexto) que deverá ser manipulada. Semelhante a isso, SourceExpression

fará menção ao nome da propriedade do objeto definido em SourceComponent, ou seja, "Text". Isto porque o Binding necessitará pegar o valor atribuído à propriedade Text do componente TEdit (edtTexto).

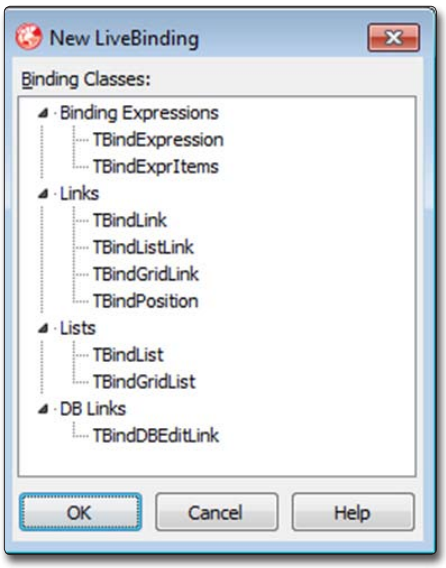


Figura 5. Tipos de LiveBinding

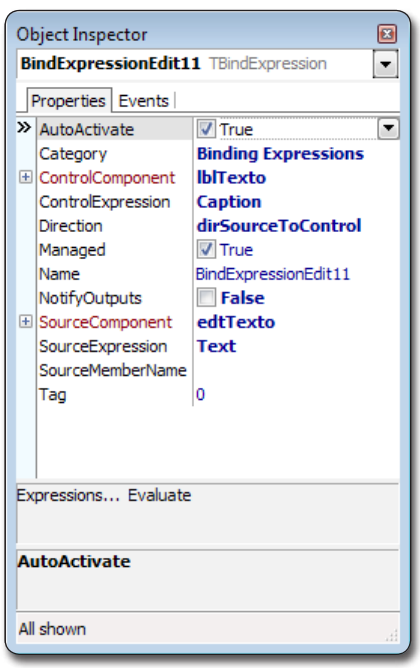


Figura 6. BindExpression

Outro item que merece destaque é a propriedade denominada Direction que diz respeito à direção do fluxo dos dados na ligação (Binding) que está sendo estabelecida. São três seus valores possíveis:

• SourceToControl: esta é a opção marcada por padrão. Ela diz que o fluxo das informações se dará do objeto origem ao destino, ou seja, de Source para Control. É o que será utilizada no exemplo;

• ControlToSource: define o fluxo inversamente à opção anterior, ou seja, as informações irão do objeto definido como Control para o objeto Source;

• Bidirectional: como o próprio nome sugere, esta opção faz um mix das outras duas. Os dados trafegam tanto de Source para Control quanto o inverso.

Uma vez definido o LiveBindings ainda se faz necessário uma última etapa para que tudo funcione conforme o esperado, que diz respeito à notificar o mecanismo de expressão (expression engine) sobre as mudanças ocorridas na expressão de entrada. Isto é necessário porque o tipo de Binding criado é gerenciado, ou seja, tem sua propriedade Managed marcada como True. Em outras palavras isso significa que o binding é avaliado automaticamente quando um objeto ou uma propriedade na expressão de entrada é modificado. De forma inversa, um binding não-gerenciado necessita ser avaliado de forma explícita e manualmente.

No nosso caso a expressão de entrada é a propriedade Text do componente edtTexto. Neste caso o evento adequado para se monitorar as mudanças ocorridas em TEdit é seu evento OnChange. O código a seguir mostra o codificação necessária para a notificação das mudanças ao mecanismo de expressão. Para isso apenas uma única linha de código é o suficiente, onde é chamado o método Notify do próprio objeto TBindingList. Este método recebe dois parâmetros, sendo o primeiro o objeto em questão e o segundo a propriedade afetada. Ainda sobre este último parâmetro, caso ele fosse passado em branco, todos os bindings que fazem referência ao objeto em questão seriam avaliados no mesmo momento.

```
procedure TfrmTestLiveBindings.edtTextoChange(Sender: TObject);
begin
 BindingsList1.Notify(edtTexto, 'Text');
end;
```

Aplicação de exemplo – FireMonkey, LiveBindings e dbExpress

A parte prática do artigo visa apresentar os passos básicos e essenciais para se construir uma aplicação FireMonkey com conectividade a banco de dados. Como já mencionado anteriormente para que isso se torne possível é imprescindível também o uso de outra tecnologia – o Live Bindings – que ficará responsável por fazer a ligação de dados entre os controles visuais e os componentes de dados. Perceba que somente com esta prática já nos atualizamos perante as novidades apresentadas pela nova versão, XE2, do Delphi, dando o start para futuros estudos sobre estas tecnologias e utilização das mesmas em projetos do seu cotidiano.

Como o intuito aqui é dar um enfoque maior na teoria e nos conceitos envolvidos no processo de desenvolvimento para este tipo de aplicação, o projeto de exemplo será bem enxuto. O mesmo possuirá uma única tela principal responsável por fazer a manutenção dos cadastros de contatos pessoais. Os dados envolvidos serão salvos um banco de dados Firebird que já dá margens para utilização de outra tecnologia, o dbExpress. A escolha deste grupo de componentes para acesso a dados se deu pelo fato da sua

grande usabilidade pela comunidade Delphi, pela presença de um driver exclusivo e nativo para o SGBD, além da praticidade em se utilizar o Data Explorer do IDE para gerência da conexão com o banco de dados.

Seguindo os passos de uma aplicação VCL tradicional a primeira etapa envolve a definição da base de dados. Pelo fato da aplicação ser trivial, o banco também é totalmente simples. Apenas para não deixar de mencionar e assim deixar o exemplo completo, a **Listagem 2** mostra o script SQL utilizado para criá-lo.

Com o banco criado, para que se aproveite os recursos do Data Explorer basta se definir uma nova conexão do tipo Firebird, conforme já mostrado aqui em artigos anteriores. Dessa forma, posteriormente, pelo simples arrastar de uma tabela apresentada no Data Explorer para, por exemplo, um Data Module, serão inseridos automaticamente os componentes de acesso a dados necessários já configurados e prontos para uso. O único complemento necessário seria a inclusão de componentes TDataSetProvider, TClientDataSet e TDataSource. Para esta demonstração mantendo o foco em facilitar ao máximo o desenvolvimento, todos eles serão colocados no próprio formulário a ser construído.

O projeto que está sendo construído será do tipo FireMonkey HD Application, que é apenas um dos quatro novos tipos de projetos voltados para FireMonkey, apresentados nessa nova versão do Delphi. A **Figura 7** exibe estas opções.

Listagem 2. Script para criação da base de dados

```
SET SQL DIALECT 3;
SET NAMES WIN1252;
SET CLIENTLIB 'fbclient.dll';

CREATE DATABASE '[caminho_banco]\DBContatos.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 8192
DEFAULT CHARACTER SET WIN1252 COLLATION WIN1252;

CREATE TABLE CONTATOS (
  CONTATO_ID  INTEGER NOT NULL,
  NOME      VARCHAR(80),
  ENDERECO   VARCHAR(80),
  CIDADE     VARCHAR(80),
  UF        CHAR(2),
  TELEFONE   VARCHAR(15)
);

ALTER TABLE CONTATOS ADD CONSTRAINT PK_CONTATOS PRIMARY KEY
 (CONTATO_ID);
```

Essencialmente o FireMonkey foi nos apresentado como sendo uma tecnologia voltada para a criação de aplicações com interfaces mais ricas e que, entre outras coisas, oferece suporte a trabalho com gráficos 3D. Justamente por isso são providos tipos específicos para estas aplicações (3D Application), conforme pôde ser visto na imagem anterior. Além disso nota-se a presença de dois tipos sufixados com iOS Application, que são providos para o desenvolvimento voltado para o sistema operacional móvel da Apple, o iOS (iPhone OS).

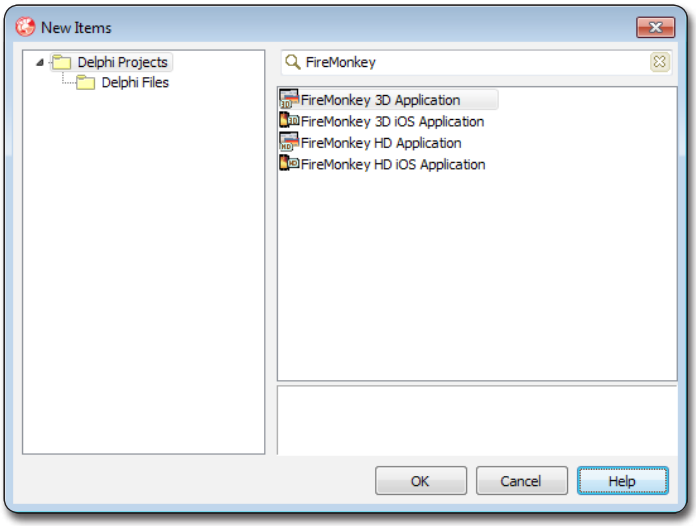


Figura 7. Novas opções de projeto, exclusivos para FireMonkey

Sendo assim FireMonkey HD Application pode ser considerado o tipo mais tradicional dentre as opções. Ao selecioná-lo temos um processo parecido com o que acontece com uma VCL Forms Application, em que a estrutura da aplicação é automaticamente criada pelo IDE, mais especificamente um projeto contendo um Form padrão já adicionado. Este já vem aberto no Form Designer para o início do desenvolvimento,

o detalhe é que enquanto na VCL este Form apresenta a extensão .dfm em FireMonkey o mesmo possui a nova extensão .fmx. Contudo, o arquivo de code-behind se mantém como .pas, com a única alteração se dando em sua seção Uses que possui agora o direcionamento para os namespaces da FMX, tais como FMX.Controls, FMX.Forms, FMX.Dialogs, ao invés de Vcl.Controls, Vcl.Forms, Vcl.Dialogs.

Um esboço da interface visual da aplicação de exemplo pode ser visto na **Figura 8**. Nela foram utilizados alguns tipos distintos de controles, como os tradicionais TLabel, TEdit e TButton que ganharam sua versão para FireMonkey, como também os exclusivos TBindNavigator e TComboEdit. Este último, aliás, é semelhante a um ComboBox comum da VCL em que podemos adicionar os itens de sua coleção através de sua propriedade Items, ao contrário do TComboBox da FMX, que não dispõe deste recurso, sendo necessário ligá-lo a uma lista. Já TBindNavigator, como o próprio nome sugere, possui as mesmas características e comportamento de um TDBNavigator, que proporciona a manipulação do registros de uma fonte de dados, vide inserir, excluir, editar, navegar, etc.

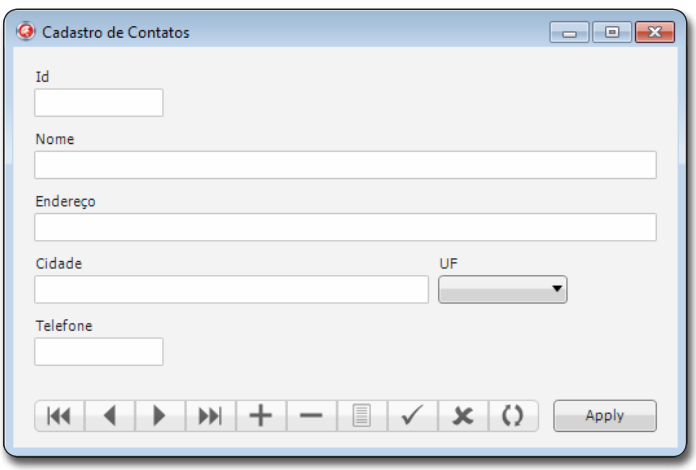


Figura 8. Interface visual da aplicação de exemplo

Seguindo os passos de um desenvolvimento padrão, se faz necessário então ligar os controles em tela com cada campo do componente de dados. Como já dito, para que estes controles se tornem data-aware será feito uso de LiveBindings. Na prática a forma mais fácil de definir um LiveBindings entre um controle visual e um campo de uma tabela é utilizando sua propriedade LiveBindings, através da opção Link to DB Field. Somente por sua nomenclatura facilmente já podemos supor que se trata de um recurso exclusivo para a conectividade necessária para o presente contexto. Neste momento pela nova caixa de diálogo que é aberta, vemos que estamos definindo um novo DB Link que se relaciona a algum dos Fields do DataSet vinculado ao DataSource. Na **Figura 9** é exibida a referida caixa de diálogo onde estão listados os campos (fields) disponíveis da tabela do banco. Portanto, este é o processo que deverá ser feito a cada um dos controles envolvidos (TEdit e TComboEdit), relativos aos campos da tabela.

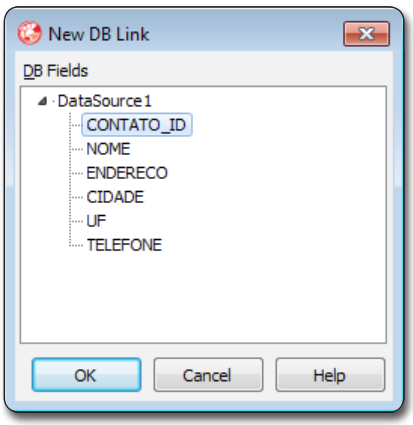


Figura 9. Definindo novo DB Link

Já na primeira definição de LiveBindings, o IDE se encarrega de adicionar automaticamente em tela outros dois componentes pertencentes ao grupo LiveBindings: TBindingList e TBindScopeDB. Em suma o primeiro, conforme já abordado anteriormente, irá centralizar e gerenciar a lista de bindings definidos, enquanto que o segundo se liga ao DataSource demarcando assim o escopo de banco de dados para o contexto, ou seja, indica a fonte de dados a ser utilizada pelo LiveBindings.

Para fixar, entenda o TBindScopeDB como um componente não-visual que provê uma forma de tornar os dados contidos em determinada fonte de dados, para qualquer componente que necessite acessá-los utilizando LiveBindings. Além disso, quando fazemos menção à TBindingsList, estamos nos referindo direta ou indiretamente ao uso de coleções de binding (ligações). Seja em aplicações VCL ou FireMonkey, cada LiveBinding criado na aplicação será encontrado em um BindingList. Tanto que ao se criar um novo LiveBinding através do ObjectInspector para determinado controle, um componente TBindingList é adicionado automaticamente no formulário em questão. Este provê ainda um editor de Binding, que pode ser acessado com um mero duplo-clique sobre o componente. A **Figura 10** exibe editor, já com todos os LiveBindings definidos para este exemplo.

Através deste editor o desenvolvedor pode visualizar as informações necessárias para uma ideal configuração e manutenção dos LiveBindings definidos na aplicação. Algumas das operações que podem ser realizadas são: criar um novo Binding ou Binding Expression, excluir um Binding existente, visualizar Binding Expressions e visualizar as propriedades de um Binding. Além disso, existe ainda um editor de expressões para LiveBindings do tipo Binding Expressions que, ao contrário dos LiveBindings do tipo DB Link, necessitam da definição de uma expressão explícita para seu funcionamento. Através dela torna-se possível a edição visual dos parâmetros de uma expressão binding. Este editor é acessado também através de um duplo-clique, em cima de um Binding Expression qualquer. Ele pode ser visto na **Figura 11**.

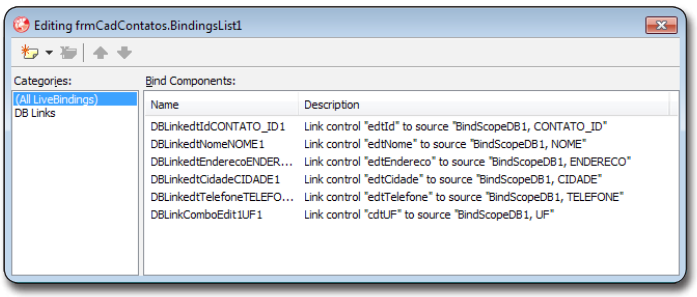


Figura 10. Editor de BindingsList

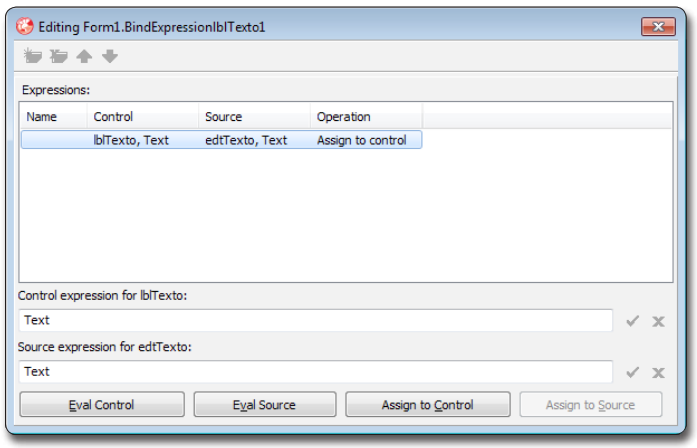


Figura 11. Editor de BindExpression

Ciente disso o controle para navegação também deverá estar ligado a esta fonte para que as operações sobre o componente de dados possam ser feitas. Em vista disso, o próprio componente TBindNavigator já provê uma propriedade denominada BindScope, onde deverá ser apontada o TBindScopeDB devido. Complementando, o botão Apply irá fazer a chamado ao método ApplyUpdates do ClientDataSet e assim salvar efetivamente as informações na tabela do banco de dados. O processo de codificação da linguagem não muda pelo fato de estarmos trabalhando com FireMonkey, portanto o código do evento ficaria da seguinte forma:

```
procedure TfrmCadContatos.btnApplyClick(Sender: TObject);
begin
  if ClientDataSet1.ApplyUpdates(0) = 0 then
    ShowMessage('Registro salvo com sucesso!');
end;
```

Em uma situação comum, numa aplicação VCL, ao invés de ShowMessage poderíamos estar chamando o método MessageBox da classe TApplication (VCL.Forms.TApplication), conforme mostrado a seguir para uma mensagem mais customizada. No entanto vale lembrar que em FMX isso ainda não é possível, uma vez que sua classe TApplication (FMX.Forms.TApplication) não provê o método MessageBox como na VCL.

```
Application.MessageBox('Registro salvo com sucesso!', 'Atenção', MB_ICONWARNING);
```

Para deixar a aplicação de exemplo totalmente funcional, a **Listagem 3** apresenta a codificação dos eventos OnCreate e OnDestroy do formulário, que irá controlar a abertura e fechamento do TClientDataSet em tela, bem como o da conexão definida no componente TSQLConnection.

Listagem 3. Codificação dos eventos Create e Destroy do Form

```
procedure TfrmCadContatos.FormCreate(Sender: TObject);
begin
  SQLConnection1.Connected := True;
  ClientDataSet1.Open;
end;

procedure TfrmCadContatos.FormDestroy(Sender: TObject);
begin
  ClientDataSet1.Close;
  SQLConnection1.Connected := False;
end;
```

Já neste momento é possível fazer os primeiros testes com a aplicação. A melhor forma de se fazer isso é testando todas as operações disponíveis, como inclusão, alteração, exclusão e navegação. Nunca se esquecendo que, por estarmos utilizando a arquitetura dbExpress em conjunto com ClientDataSet, para se efetivar qualquer alteração deve-se sempre confirmar a operação através do botão Apply. Do contrário, as mudanças serão persistidas somente em memória, sendo perdidas ao finalizar o aplicativo. A **Figura 12** mostra o exemplo já em execução, com dados sendo devidamente salvos no banco de dados.

Estilizando uma aplicação FireMonkey

Em aplicações Web quando se menciona a estilização de uma página HTML, está se fazendo referência à definição da apresentação dos documentos escritos nesta linguagem. Em outras palavras, define a forma visual de como a página será apresentada ao usuário. Para isso, comumente é utilizado um recurso denominado CSS (Cascading Style Sheets), que é uma linguagem de estilo utilizada exclusivamente para esta finalidade, colocando-se assim toda a formatação em um único documento a parte (.css), isolado da página HTML em si, mas vinculado à mesma. Em aplicações FireMonkey isto também é possível, e similarmente aos arquivos .css temos arquivos .style, que definem a apresentação visual da aplicação.

Uma aplicação FMX é composta por um conjunto de controles FireMonkey, os quais são compostos basicamente de sub-controles e formas primitivas, sendo lapidados por efeitos que tornam seu visual mais rico. Desta forma no FireMonkey os estilos são definidos a partir desta composição de recursos e armazenados em uma estrutura denominada style book (livro de estilos). Cada classe de controle FMX possui nativamente um estilo padrão, definido por código (hard coded). Isso pode ser visto em qualquer controle através do menu de contexto do objeto pela opção Edit Default Style. Ao fazer isso uma nova janela é aberta, denominada Style Designer, aonde o IDE exibe uma cópia exata do código interno que define o estilo. Exemplificando, o estilo padrão de um TEdit qualquer contido na aplicação de exemplo é mostrado na **Figura 13**.

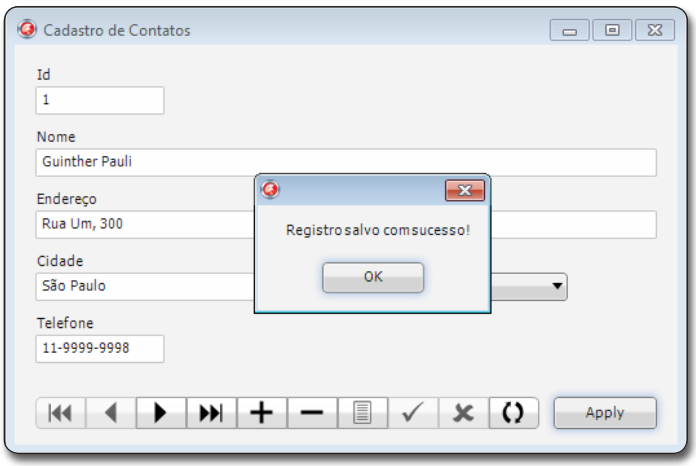


Figura 12. Aplicação de exemplo em execução

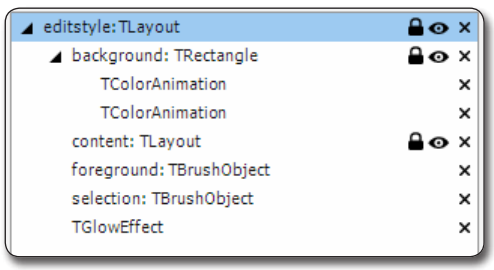


Figura 13. Estilo padrão de um TEdit

Este estilo aparece nomeado como "editstyle", referenciando a classe TLayout. Esta linha representa efetivamente o controle em si. Ainda sobre a estrutura apresentada, mesmo se tratando de um simples componente sua composição se mostra um tanto quanto complexa em vista de seus sub-elementos de variados tipos. Fazendo menção a alguns destes sub-elementos, o de nomeação "background" faz referência a um TRectangle, que dá forma à aparência retangular de fundo do controle. Conforme pode ser visto, ele contém dois elementos internos do tipo TColorAnimation, responsáveis pela animação de cores disparada pelo passar do mouse sobre o controle, causando um efeito visualmente agradável. Seguindo essa mesma linha de raciocínio os demais elementos são auto-denominativos, como "content" que define a área de conteúdo do controle e "selection" que determina a cor a ser utilizada pelo controle no momento de sua seleção.

A própria instalação do IDE já traz nativamente alguns destes arquivos de estilo, prontos para serem utilizados. Por defaul, eles ficam localizados na pasta de documentos públicos do Rad Studio, denominada Styles (C:\Users\Public\Documents\RAD Studio\9.0\Styles, em uma instalação em inglês). Apenas para citar, a própria nomenclatura destes arquivos dão uma idéia do que se pode esperar em termos visuais, tais como FMX.Platform.Mac.style, FMX.Platform.iOS.style e FMX.Platform.Win.style.

Vendo isso na prática iremos então adicionar o recurso de estilização em nossa aplicação de exemplo, que até o momento já está

totalmente funcional. Em termos visuais apenas será necessário o acréscimo de um TButton. Será através dele que o usuário poderá então selecionar um novo estilo a ser aplicado instantaneamente na aplicação. Como será necessária uma seleção de arquivo, isso pode ser feito através do uso de um componente TOpenDialog já direcionado para o tipo de arquivo do estilo (.style), e apontado para o caminho padrão onde estão localizados. A seguir, é exibida a configuração das propriedades do objeto.

```
object OpenDialog1: TOpenDialog
  DefaultExt = '*.style'
  FileName = '*.style'
  Filter = 'Style(*.style)'
  InitialDir = 'C:\Users\Public\Documents\RAD Studio\9.0\Styles'
...
```

Por fim, o diferencial nesta etapa se faz pela adição de um novo componente exclusivo às aplicações FireMonkey, denominado TStyleBook. Ele será o responsável por carregar o arquivo de estilo a ser aplicado na aplicação, podendo assim prover uma coleção de estilos ao formulário. Cada objeto TForm, por sinal, possui uma propriedade StyleBook, que recebe um objeto do tipo TStyleBook. Portanto, com o acréscimo de um objeto deste ao formulário de cadastro do projeto, a codificação do evento OnClick do botão deverá ser a apresentada na **Listagem 4**.

Dentre as propriedades do componente TStyleBook a que interessa neste ponto é a FileName, que deverá receber o nome do arquivo de estilo, conforme mostra a listagem apresentada. A **Figura 14** mostra a aplicação de exemplo estilizada com o arquivo FMX.Platform.iOS.style. A diferença visual é perceptível, em que alguns detalhes podem ser notados, como os cantos arredondados dos controles e as cores dos botões.

Ainda sobre o componente TStyleBook, um Form pode possuir N objetos deste tipo, no entanto, sua propriedade StyleBook pode fazer referência a somente um por vez. Esta propriedade nativamente apresenta-se sem valor algum definido, fazendo com que o Form faça uso do seu estilo padrão. Sendo assim, temos uma maneira facilitada de "limpar" qualquer estilização que tenha sido aplicada. Colocando esta ação em um botão, poderia se usar uma codificação semelhante à **Listagem 5**, onde é atribuído nil à propriedade StyleBook do formulário de cadastro da aplicação de exemplo.

Conclusão

Desde o lançamento da recente versão do Delphi, a XE2, estamos cada vez mais acompanhando suas inúmeras novidades, principalmente através da Web e de palestras, através dos eventos oficiais realizados pela própria Embarcadero. Ao mesmo tempo, a Clube Delphi mantém o compromisso em trazer ao leitor tudo o que está acontecendo no meio Delphi, tornando-se também uma fonte de informação à comunidade. O primeiro passo foi dado com o artigo que apresentou um overview das principais novidades dessa nova versão do IDE. A partir dele, abordagens dedicadas a assuntos específicos serão os temas de artigos futuros, assim como o presente artigo, onde o foco foi o FireMonkey.

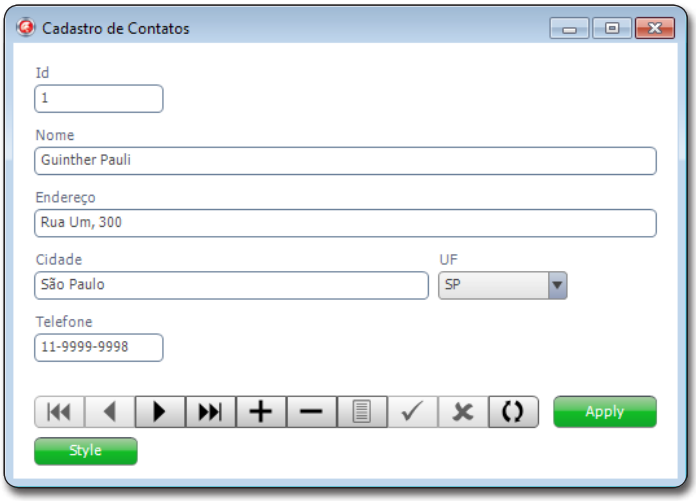


Figura 14. Aplicação de exemplo em execução e estilizada

Listagem 4. Codificação do evento OnClick do botão referente a Style

```
procedure TfrmCadContatos.btnStyleClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
    StyleBook1.FileName := OpenDialog1.FileName;
end;
```

Listagem 5. Codificação do evento OnClick do botão para se voltar à estilização padrão

```
procedure TfrmCadContatos.btnLimparEstiloClick(Sender: TObject);
begin
  Self.StyleBook := nil;
end;
```

O FireMonkey (ou FMX), assim como o LiveBindings, certamente foram duas das novidades que causaram mais frisson na comunidade, muito antes mesmo do lançamento efetivo da versão XE2, ainda no momento de seu anúncio. Isto se justifica pelo fato de ambos serem recursos relativos a áreas até então inexploradas pelo Delphi: criação de interfaces de usuário visualmente mais ricas e multi-plataforma e ligação de dados entre objetos através de expressões. Dessa forma, somente com esta premissa, os desenvolvedores puderam enxergar novas possibilidades de estudos e de negócios, causando uma expectativa enorme e inédita. Com a chegada do Delphi XE2, a eficácia destes elementos pôde enfim ser comprovada, e aos poucos os projetos atuais vão ganhando o acréscimo de tais recursos.

Assim sendo, este artigo procurou explorar um pouco mais a fundo esta mais recente plataforma de desenvolvimento da Embarcadero, o FireMonkey, culminando num exemplo prático onde foi exposto o seu uso com banco de dados. Em função disso, pôde ser vista a necessidade do uso de Live Bindings para a ligação entre controles visuais e componentes

de dados, definindo assim a proximidade existente entre as duas novas tecnologias. Outro ponto a ser mencionado é que por se tratar de uma nova forma de desenvolvimento, o artigo teve a preocupação em traçar sempre que possível, paralelos com o desenvolvimento tradicional com a VCL. Indo um pouco além, a partir do que foi mostrado, fica fácil você começar a construir suas primeiras aplicações em FireMonkey, com o acréscimo de dbExpress e LiveBindings.

Delphi XE2 Architect - Trial
https://downloads.embarcadero.com/free/delphi

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 5 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas além de participar do programa Beta Tester Delphi Prism, da Embarcadero.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

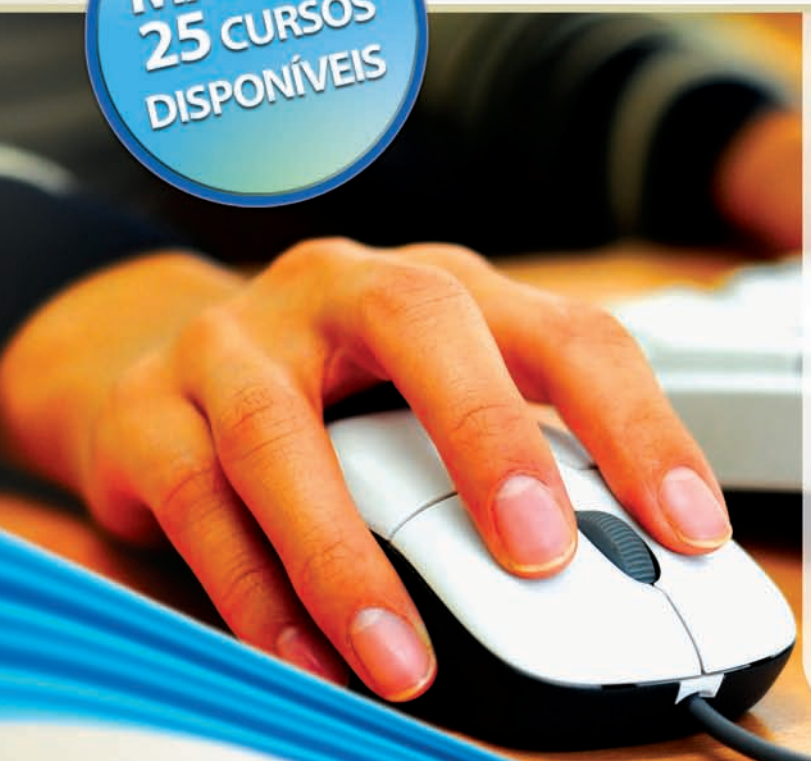
www.devmedia.com.br/clubedelphi/feedback

CURSOS ONLINE

.net magazine

A Revista .net Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

MAIS de 25 CURSOS DISPONÍVEIS



CONHEÇA OS CURSOS MAIS RECENTES:

- Curso .net Entity Framework - Loja Virtual
- Novidades do Visual Studio 2010
- Componentes DevExpress
- Crie um chat utilizando Asp.Net 3.5 e Linq
- Novidades do C# (csharp) 3.0
- POO com C#
- Microsoft Visual Studio Tools for Office (VSTO)
- Computação na Nuvem com a Plataforma Microsoft Windows Azure

Para mais informações :

www.devmedia.com.br/curso/netmagazine

(21) 3382-5038

http://www.devmedia.com.br

DevMedia

Aprofundando em Int
Teoria e Prática

Resumo DevMan

De que trata o artigo:

O artigo trata de interfaces, uma maneira de se estabelecer um "contrato" entre duas entidades, sejam elas objetos, componentes ou outros programas, de modo que possam se comunicar através de trocas de mensagens que nada mais são do que chamadas de métodos.

Em que situação o tema é útil:

Sempre que for necessário escrever alguma classe que siga um comportamento padrão sem que necessariamente seja de uma linhagem específica. Com interfaces é possível, por exemplo, escrever um método que aceite um objeto que seja de uma interface, e que outro programador que escreverá e instaciará este objeto. Qualquer tipo de objeto pode ser passado contanto que implemente esta interface. Outra possibilidade é especificar um comportamento padrão, um conjunto de métodos, sem se preocupar em como eles serão implementados.

Aprofundando em Interfaces:

Se você já programa orientado a objetos a algum tempo, está familiarizado com interfaces. Para quem já viu o .Net framework ou o Java, o uso que se faz de interfaces nesses ambientes é enorme. Um exemplo disso é que no .Net framework para se criar um objeto do tipo "coleção" basta implementar certas interfaces não importando como a coleção de objetos é implementada internamente, se é vetor, lista ligada ou outro tipo de coleção. A mesma coisa vale para tipos lista, command, connection e assim por diante. No Delphi também podemos fazer isso: criando vários objetos que implementem a mesma interface dá a possibilidade de armazenar qualquer um deles numa variável do tipo dessa interface.

erfaces

Vitor Luiz Rubio

No artigo anterior, Interfaces – Teoria e Prática, foi explorada a teoria sobre interfaces. Foram apresentadas características das mesmas, como declará-las e usá-las em Delphi e ainda foi mostrado sobre as GUIDs.

Também foi apresentada a agregação, mostrando como um objeto pode implementar várias interfaces simulando herança múltipla ou como um objeto pode implementar uma interface e conter uma referência para outro que implemente outra interface. Foram feitas algumas experiências com veículos e animais. Outro ponto interessante para ser mostrado sobre as interfaces é a contagem de referências. Neste artigo daremos continuidade ao assunto explorando sobre como utilizar herança com interfaces e como manter referências de forma correta.

Interfaces com propriedades

Considere-se a interface ICliente da **Listagem 1.**

Interfaces podem ter propriedades, contanto que elas estejam configuradas com métodos acessores que também façam parte da interface. Como consequência, os métodos acessores deverão ser públicos na implementação da interface, pois todos os métodos de uma interface devem ser implementados como públicos.

Isso não é uma consequência ruim. Os métodos acessores públicos podem ser usados de igual forma ao Java ou C++: invocando o acessor Set para mudar o valor de um atributo ou o acessor Get para obter o valor do atributo. Mesmo assim, há vantagem de usar uma propriedade diretamente em vez dos assessores. Propriedades que escrevem ou leem seus valores diretamente nos campos privados podem constituir uma violação de encapsulamento. Entretanto, em alguns casos, permite-se a leitura direta e a escrita usando-se um método acessor. Cada caso é um caso e a própria VCL tem muitos exemplos de classes e componentes com propriedades que leem ou escrevem diretamente de campos privados.

Também não é nenhum crime expor os campos se eles forem realmente atributos relevantes da entidade que precisam ser vistos e modificados pelo ambiente externo, contanto que não haja uma maneira especial de se ler ou gravar esses atributos.

Sempre que houver uma maneira especial de se ler ou gravar esses atributos, como cálculos, validações ou máscaras, sempre é possível encapsular este atributo com métodos acessores e vincular esses acessores a uma propriedade, permanecendo o acesso ao atributo transparente.

Nota do DevMan

Métodos acessores são métodos usados para se ler ou modificar o valor de um atributo, sendo que o atributo pode permanecer privado e ser exposto através de métodos. É uma forma de encapsulamento do atributo/propriedade.

Geralmente métodos acessores para se ler um atributo começam com "Get" e para modificar o atributo começam com Set.

Em linguagens como Java e C++ é muito comum ter-se vários atributos privados e um método Get ou Set para cada um que necessitar ser publicado ou exposto para o ambiente externo.

Outras linguagens, como o Delphi, permitem que métodos acessores (privados ou públicos) sejam automaticamente ativados por propriedades.

A vantagem disso é que propriedades podem ser usadas como variáveis, estando de qualquer um dos lados de uma atribuição ou expressão, usadas com operadores +, -, /, := e assim por diante, coisa que não dá para ser feita com métodos sem o uso de parâmetros ou que estes estejam à direita das expressões.

Adicionalmente, linguagens como C# no .Net Framework criam em linguagem intermediária métodos acessores automaticamente se estes não forem especificados no escopo de uma propriedade.

Listagem 1. Interface ICliente

```
unit uICliente;

interface

type
 ICliente = interface
 ['{3A192476-2CF2-4895-8B97-7DED7A8EC181}']
   procedure SetNome(const vNome: string);
   procedure SetCPF(const vCPF: string);
   procedure SetDtNasc(const vDtNasc: TDateTime);
   procedure SetEndereco(const vEndereco: string);

   function GetNome: string;
   function GetCPF: string;
   function GetDtNasc: TDateTime;
   function GetEndereco: string;

   property Nome: string read GetNome write SetNome;
   property CPF: string read GetCPF write SetCPF;
   property DtNasc: TDateTime read GetDtNasc write SetDtNasc;
   property Endereco: string read GetEndereco write SetEndereco;
 end;

implementation

end.
```

Nota

O código completo está disponível para download, no artigo são apresentados apenas trechos do código.

Implementação de interfaces

A interface ICliente da **Listagem 1** contém uma propriedade chamada CPF dando a entender que o cliente sempre será uma pessoa física. Isso poderia ser feito de outra forma, a propriedade poderia se chamar Documento e classes diferentes de pessoa física ou jurídica implementariam esta mesma interface. Uma classe que pode implementar a interface ICliente é a classe TCliente da **Listagem 2**.

Listagem 2. Classe TCliente

```
type
 TCliente = class(TInterfacedObject, ICliente)
 private
  FNome: string;
  FEndereco: string;
  FCPF: string;
  FDtNasc: TDateTime;
 public
  procedure SetNome(const vNome: string);
  procedure SetCPF(const vCPF: string);
  procedure SetDtNasc(const vDtNasc: TDateTime);
  procedure SetEndereco(const vEndereco: string);
  function GetNome: string;
  function GetCPF: string;
  function GetDtNasc: TDateTime;
  function GetEndereco: string;
  property Nome: string read GetNome write SetNome;
  property CPF: string read GetCPF write SetCPF;
  property DtNasc: TDateTime read GetDtNasc write SetDtNasc;
  property Endereco: string read GetEndereco write SetEndereco;
 end;
```

O objetivo de se implementar clientes através de interfaces é lidar com tipos diferentes de clientes. Não só o clássico "pessoa física" e "pessoa jurídica" mas clientes especiais como "cinco estrelas", "mensalistas", "diaristas" e assim por diante. Essa categorização pode variar de acordo com sua regra de negócio. Imaginando que um cliente, além de ser pessoa física ou jurídica, pode ser estrangeiro, ao utilizar listas genéricas de clientes podemos ter um parâmetro que define o tipo de cliente que elas usam.

Interfaces com Generics

Interfaces também podem ter parâmetros genéricos ou servir como parâmetros desse tipo para outra classe. Isso acrescenta dois níveis de abstração no modelo, pois além do uso de interfaces permitir que várias classes a implementem de uma forma diferente, o uso de um parâmetro genérico faz com que se crie várias interfaces baseadas na mesma. Uma interface genérica é

declarada de forma muito similar à não genérica, como mostra o código a seguir.

```
IInterfaceGenerica<T> = interface
```

E para que classes possam implementar essa interface, elas devem informar qual o tipo do parâmetro genérico T, como é visto a seguir.

```
TClasseGenericaInteiro = class(TInterfacedObject, IInterfaceGenerica<Integer>)
 public
  function Soma(a, b: Integer): Integer;
  function Subtracao(a, b: Integer): Integer;
 end;
```

Para entender o uso dos parâmetros genéricos vamos analisar a interface IInterfaceGenerica<T>. Essa interface expoe alguns métodos que representam operações matemáticas. No caso da classe TClasseGenericaInteiro essas operações são realizadas com números inteiros. Isso é definido ao passar na declaração da classe o tipo Integer. Outras classes foram criadas empregando outros tipos, como String e Extended, ou seja, a mesma operação matemática só que envolvendo tipos diferentes. Imaginando um uso dessas classes, teríamos um formulário que instaciaria cada classe dessas e lançaria em um memo o resultado de suas operações matemáticas envolvendo os diversos tipos, exatamente como mostra a **Listagem 3**. Na **Figura 1** vemos o resultado.

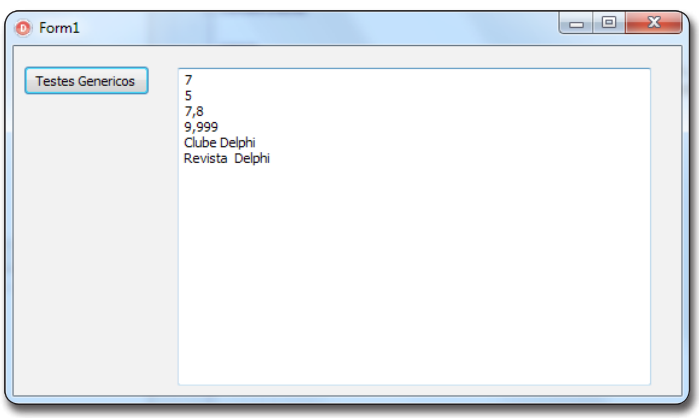


Figura 1. Testes com interfaces genéricas

O que pode ser observado é que IInterfaceGenerica<Integer> é diferente de IInterfaceGenerica<Extended>, que é diferente de IInterfaceGenerica<String>. Por isso, as variáveis iinteiro, iextended e istring não puderam ser declaradas como sendo do mesmo tipo, e não podem ser intercambiadas entre si. Entretanto, caso houvesse alguma outra classe que implementasse IInterfaceGenerica<String> ela poderia ser intercambiada.

Por exemplo, pode-se ter uma outra classe que implemente IInterfaceGenerica<string> de uma forma um pouco diferente, como é mostrado a seguir e assim ela pode ser intercambiada com a primeira, como mostra a **Listagem 4**.

```
TOutraClasseGenericaString = class(TInterfacedObject, IInterfaceGenerica<string>)
 public
  function Soma(a, b: string): string;
  function Subtracao(a, b: string): string;
 end;
```

Outra possibilidade é uma classe genérica implementar uma interface genérica, ou uma classe genérica implementar uma interface comum. Considerando a seguinte classe:

```
TClasseGenerica<T> = class(TInterfacedObject, IInterfaceGenerica<T>)
 public
  function Soma(a, b: T): T;
  function Subtracao(a, b: T): T;
 end;
```

Listagem 3. Modo de usar

```
procedure TForm1.Button1Click(Sender: TObject);
var
 iinteiro: TClasseGenericaInteiro;
 iextended: TClasseGenericaExtended;
 istring: TClasseGenericaString;
begin

 iinteiro := TClasseGenericaInteiro.Create;
 iextended := TClasseGenericaExtended.Create;
 istring := TClasseGenericaString.Create;

 Memo1.Lines.Add( inttostr(iinteiro.Soma(3,4)) ); //7
 Memo1.Lines.Add( inttostr(iinteiro.Subtracao(9,4)) ); //5

 Memo1.Lines.Add( FloatToStr(iextended.Soma(3.2,4.6)) ); //7.8
 Memo1.Lines.Add( FloatToStr(iextended.Subtracao(10,0.001)) ); //9.999

 Memo1.Lines.Add( istring.Soma('Clube','Delphi') );
 Memo1.Lines.Add( istring.Subtracao('Revista Clube Delphi','Clube') );

end;
```

Listagem 4. Modo de usar a outra classe

```
procedure TForm1.Button1Click(Sender: TObject);
var
 iinteiro: IInterfaceGenerica<Integer>;
 iextended: IInterfaceGenerica<Extended>;
 istring, iOutraString: IInterfaceGenerica<string>;
begin

 iinteiro := TClasseGenericaInteiro.Create;
 iextended := TClasseGenericaExtended.Create;
 istring := TClasseGenericaString.Create;
 //aqui é onde ocorre o intercambio de classes
 iOutraString := TOutraClasseGenericaString.Create;

 Memo1.Lines.Add( inttostr(iinteiro.Soma(3,4)) ); //7
 Memo1.Lines.Add( inttostr(iinteiro.Subtracao(9,4)) ); //5

 Memo1.Lines.Add( FloatToStr(iextended.Soma(3.2,4.6)) ); //7.8
 Memo1.Lines.Add( FloatToStr(iextended.Subtracao(10,0.001)) ); //9.999

 Memo1.Lines.Add( istring.Soma('Clube','Delphi') );
 Memo1.Lines.Add( istring.Subtracao('Revista Clube Delphi','Clube') );

 Memo1.Lines.Add( iOutraString.Soma('Teste','Teste') );
 Memo1.Lines.Add( iOutraString.Subtracao('Teste','Teste') );

end;
```

Não seria possível saber como retornar a soma ou a subtração de T sem saber o que é T e se T é passível de ser usada com as operações de soma ou subtração, ou ainda, se T fosse de algum tipo ou classe que pudesse ter um valor ou em que pudesse ser aplicado um operador isso seria possível.

Uma maneira de fazer isso é criar uma classe ou interface que implemente sobrecarga de operadores e usar constraints para definir como a soma ou a subtração pode ser feita.

Generics definem o que uma classe "É", enquanto que interfaces dizem o que uma classe "FAZ", lembrando que, o que uma classe faz (seu comportamento) é o que mais importa em certas situações e diz muito sobre o que uma classe "PODE SER".

Tenha em mente que um parâmetro genérico diz de que tipo os retornos e parâmetros de métodos deve ser, podendo fazer com que uma mesma classe possa ser reproduzida em classes que tenham métodos com assinaturas diferentes. Já a interface faz o contrário: ela permite que classes diferentes tenham métodos com assinatura igual.

Uma classe genérica implementar uma interface comum é mais trivial. O exemplo da **Listagem 5** mostra isso.

Listagem 5. Classes genéricas e interfaces comuns

```
TClasseGenericaComInterfaceComum<T> = class(TInterfacedObject,
InterfaceComum)
public
  procedure UmMetodo;
end;

TClasseComumComInterfaceComum = class(TInterfacedObject, InterfaceCo-
mum)
public
  procedure UmMetodo;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  itf1, itf2: InterfaceComum;
begin

  itf1 := TClasseGenericaComInterfaceComum<integer>.Create;
  itf2 := TClasseComumComInterfaceComum.Create;

  itf1.UmMetodo;
  itf2.UmMetodo;

end;
```

Temos duas classes, uma comum e uma com parâmetro genérico, que podem implementar uma mesma interface. Os objetos itf1 e itf2 são do mesmo tipo, InterfaceComum, mas recebem objetos de classes diferentes que implementam InterfaceComum. As duas variáveis tem o método "UmMetodo" invocado. Isso é possível porque o "UmMetodo" não usa o parâmetro genérico.

Classes e interfaces genéricas não podem ser usadas sem o parâmetro genérico e são de tipos diferentes para cada parâmetro. Uma maneira onde classes genéricas podem ser usadas junto com interfaces é em listas e coleções.

A classe TList<T> aceita parâmetros genéricos de qualquer tipo, já a classe TObjectList<T> só aceita objetos, pois foi declarada com uma constraint do tipo class:

```
TObjectList<T: class> = class(TList<T>)
```

Interessante é o fato de que uma classe genérica pode herdar de outra classe genérica modificando a constraint do parâmetro genérico, então ela pode fazer o mesmo com uma interface. Sendo assim, um carrinho de compras poderia ser um TCarrinho<T: IItemVenda> = class(TList<T>). Veja o exemplo da **Listagem 6**.

Listagem 6. Listas que aceitam objetos de certa interface

```
TListaDeInterfaceComum<T: InterfaceComum> = class(TList<T>
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  itf1, itf2: InterfaceComum;
  Lista1: TListaDeInterfaceComum<InterfaceComum>;
  Lista2: TListaDeInterfaceComum<TClasseComumComInterfaceComum>;
  Lista3: TListaDeInterfaceComum<TClasseGenericaComInterfaceComum
  <Integer>>;
begin

  itf1 := TClasseGenericaComInterfaceComum<integer>.Create;
  itf2 := TClasseComumComInterfaceComum.Create;

  itf1.UmMetodo;
  itf2.UmMetodo;

  Lista1 := TListaDeInterfaceComum<InterfaceComum>.Create;
  Lista2 := TListaDeInterfaceComum<TClasseComumComInterfaceComum>.Create;
  Lista3 := TListaDeInterfaceComum<TClasseGenericaComInterfaceComum
  <Integer>>.Create;

  Lista1.Add(itf1);
  Lista1.Add(itf2);

  Lista2.Add(TClasseComumComInterfaceComum.Create);

  Lista3.Add(TClasseGenericaComInterfaceComum<integer>.Create);

end;
```

A lista 1 do tipo TListaDeInterfaceComum<InterfaceComum> aceita qualquer um dos objetos itf1 ou itf2. Já as listas 2 e 3 aceitam apenas parâmetros do tipo TclasseComumComInterfaceComum e TClasseGenericaComInterfaceComum<Integer>> e não vão aceitar as variáveis itf1 ou itf2 mesmo que sejam de uma interface suportada.

Como TListaDeInterfaceComum só aceita parâmetros genéricos do tipo InterfaceComum as construções

```
Lista1: TListaDeInterfaceComum<InterfaceComum>;
Lista2: TListaDeInterfaceComum<TClasseComumComInterfaceComum>;
Lista3: TListaDeInterfaceComum<TClasseGenericaComInterfaceComum<Integer>>;
```

São válidas, mas a construção Lista4: TListaDeInterfaceComum <string> não é válida, retornando o erro de compilação "[DCC Error] uTeoria.pas(63): E2514 Type parameter 'T' must support interface 'InterfaceComum'", pois é obvio que o tipo string passado no parâmetro genérico não suporta o tipo InterfaceComum.

Essa última listagem mostra que os parâmetros genéricos podem servir para restringir que tipo de objetos que uma lista recebe, caso for um parâmetro muito específico. Mas pode servir para tornar o parâmetro abrangente ao mesmo tempo que só aceite objetos que implementem uma certa interface. Por exemplo uma TList<TItemProduto> é uma lista que aceitaria apenas itens do tipo produto, enquanto que uma TList<IItemVenda> não aceitaria objetos de apenas um tipo (ou seus descendentes), mas qualquer objeto que implementasse IItemVenda, sendo produto ou serviço.

Colocando em prática

Considere-se uma empresa de customização de automóveis que tanto vende autopeças como as instala, prestando assim serviços. Quando o sistema dessa empresa imprime um pedido, este tem produtos e serviços. Pode ter também pagamento de mão de obra terceirizada, pagamentos de taxas e assim por diante. A nota fiscal ou nota fiscal eletrônica dessa empresa também é uma de produtos e serviços.

Muitos programadores colocam produtos e serviços todos em uma mesma tabela de banco de dados separados por algumas flags. Isso não é errado em si, mas é ruim porque todos os campos específicos de produtos ficam null em registros de serviços e vice-versa. Porém, a persistência não é o foco desse artigo.

Interfaces são ideais para criar objetos compatíveis e intercambiáveis, sem que sejam necessariamente da mesma linhagem. Não importa o tamanho do projeto pois seu uso favorece um design de baixo acoplamento porque estaremos programando para abstrações e não classes concretas. Mas não é apenas isso, interfaces servem de pontos de encaixe para plugins pois seus métodos serão chamados independentemente de sua implementação.

O objetivo do sistema é juntar em um pedido o total de produtos e de serviços, mas mostrar o subtotal de cada um. Como exercício pode-se extrair a comissão dos vendedores e a instalação dos técnicos.

Para tanto, será criada uma interface IPedido que conterá a lista dos itens, o total, o subtotal de produto, o subtotal de serviços e qualquer outro cálculo fiscal ou tributário que seja necessário. Não é o objetivo deste artigo criar um sistema de ponto de venda ou nota fiscal completo, mas apenas mostrar onde pode-se usar interfaces nos softwares mais comuns de se criar hoje em dia.

Há certas similaridades entre um pedido, um orçamento e uma nota fiscal: um orçamento é quase idêntico a um pedido, com todos os campos e totais, mas ele não é efetivo, não conta como uma entrada ou crédito no seu fluxo de caixa e não gera uma saída do estoque. Já uma nota fiscal contém todas as informações do pedido e mais algumas outras. Algumas dessas informações são replicadas ou desnormalizadas por exigências da legislação.

Muitos programadores transformam pedidos e orçamentos em uma única tabela no banco de dados com uma flag dizendo se é um pedido efetivo ou não. E para emitir a nota fiscal usam duas tabelas a parte, uma de itens e uma de nota fiscal, para replicar esses dados. É possível perceber aqui que essas três entidades: pedido, orçamento e nota fiscal implementam uma interface em comum. Mais do que isso, para um orçamento se tornar um pedido efetivo basta apenas que se "migre" uma pequena informação.

Usando um pouco de abstração pode-se imaginar que em qualquer mercado ou loja pode-se colocar os produtos em um carrinho ou cesta e tomar uma de três ações:

1 – Largar o carrinho sem passar no caixa → carrinho abandonado
2 – Passar no caixa, mas desistir da compra → nesse caso ele chega a ver o total, houve um orçamento
3 – Efetuar a compra → nesse caso é decidido o meio de pagamento

Em e-commerce usa-se muito essa metáfora do carrinho ou cesta de compras para se tirar estatísticas dos carrinhos abandonados. Esse objeto carrinho que dá origem tanto a pedidos efetivos como orçamentos ou notas fiscais pode ser usado em sistemas desktop também.

Um carrinho não é uma mera coleção de itens, mas ele pode fazer cálculos com esses itens. Objetos do tipo TOrcamento, TPedido e TNotaFiscal podem implementar a interface de ICarrinho diretamente ou delegar essa tarefa a uma instância de TCarrinho.

Como carrinho é uma coleção, pode ser implementado através de generics. Os itens dessa coleção podem ser produtos ou serviços, então é possível abstrair uma interface entre esses dois.

Outro benefício do uso de interfaces na prática é a redução de "ifs" e "cases" desnecessários. Não que desvios condicionais sejam ruins, afinal eles são a base da abstração e são eles que diferenciam um "programa" de um "batch", mas sempre que ifs ou cases são usados para se saber o "tipo" de um item (ou o valor de uma propriedade ou flag qualquer) para se fazer cálculos de maneira diferenciada, então há um sério problema no seu código: sempre que esses cálculos precisarem ser refeitos, as mesmas estruturas if ou case serão repetidas, e sabemos que um dos princípios da boa programação é a sigla em inglês DRY (don't repeat yourself, que significa "não se repita"), ou seja, não repetir o mesmo trecho de código ou lógica várias vezes.

Para ilustrar o quanto pode ser complicado dar manutenção em um sistema sem o uso de interfaces (e sem outros bons conceitos da POO), este artigo mostrará uma pequena parte desse hipotético sistema de customização de automóveis: apenas a parte de se adicionar itens em um pedido. Depois será mostrado o mesmo sistema com o uso de interfaces. As evoluções necessárias serão feitas apenas no código que usa interfaces, por ser mais fácil.

Como não é escopo deste artigo, não foram criadas as classes de Produto e Serviço, com as descrições e outras informações. Tenha em mente no entanto que um item de venda (esteja ele em um carrinho, orçamento ou pedido) sempre tem um produto ou serviço associado. Indo mais a fundo, produto e serviço também são duas classes que podem ser completamente distintas, mas ter uma interface em comum.

Nota

Todos os exemplos deste artigo são compilados com a opção ReportMemoryLeaksOnShutdown := true;no arquivo .dpr.Isso é necessário e altamente recomendável pois o Delphi não tem um Garbage Collector como o ambiente .Net ou Java, e cabe ao programador a responsabilidade de eliminar objetos da memória depois de utilizados. O uso de interfaces pode minimizar o vazamento de memória decorrente de descuidos, mas o uso errado de interfaces pode causar vazamentos de memória bem mais difíceis de se encontrar.

A **Listagem 7** mostra uma classe TItemVenda que não usa interfaces. A **Listagem 8** mostra um carrinho que implementa uma coleção de itens de venda através de uma lista genérica e que pode ser usada como base para pedidos, orçamentos etc. A **Figura 2** é um formulário comum, com um StringGrid, que exibe os produtos ou serviços adicionados ao pedido e mostra o total geral. A **Listagem 9** mostra parte do código deste formulário.

Listagem 7. Item de venda sem uso de interfaces

```
type
 TTipoItem  = (tpProduto, tpServico);
 TItemVenda = class
 private
  FTipo: TTipoItem;

  FValorUnitario: Double;
  FQuantidade: Integer;

 public
  constructor Create(tipo: TTipoItem; unitario: Double; qtde: Integer);
  function GetTotal: double;
  property ValorUnitario: Double read FValorUnitario write FValorUnitario;
  property Quantidade: Integer read FQuantidade write FQuantidade;
  property Tipo: TTipoItem read FTipo write FTipo;
 end;
```

Listagem 8. Carrinho sem o uso de interfaces

```
type
 TCarrinho = class
 private
  FLista: TList<TItemVenda>;
 public
  constructor Create;
  destructor Destroy; override;
  procedure Adicionar(const item: TItemVenda);
  procedure Remover(const item: TItemVenda); overload;
  procedure Remover(const idx: integer); overload;
  function GetItem(idx: integer): TItemVenda;
  procedure SetItem(idx: integer; const item: TItemVenda);
  function GetTotal: double;
  function TotalItens: integer;

  property Itens[idx: integer]: TItemVenda read GetItem write SetItem;
 end;
```

Listagem 9. GUI do pedido sem uso de interfaces

```
procedure TForm2.Apresenta(carrinho: TCarrinho);
var i: integer;
begin

 StringGrid1.RowCount := carrinho.TotalItens + 1; //o +1 é o cabeçalho
 for i := 0 to carrinho.TotalItens -1 do
 begin
  StringGrid1.Cells[0, i+ 1] := FloatToStr( carrinho.Itens[i].ValorUnitario );
  StringGrid1.Cells[1, i+ 1] := IntToStr( carrinho.Itens[i].Quantidade );
  StringGrid1.Cells[2, i+ 1] := FloatToStr( carrinho.Itens[i].GetTotal );
 end;

 edTotalGeral.Text := FloatToStr( carrinho.GetTotal );
end;

procedure TForm2.btAdicionarTesteClick(Sender: TObject);
begin
 FCarrinho.Adicionar( TItemVenda.Create(tpProduto, 15, 2) ); //30
 FCarrinho.Adicionar( TItemVenda.Create(tpProduto,13, 5) ); //65
 FCarrinho.Adicionar( TItemVenda.Create(tpServico, 100, 0) );  //100

 Apresenta(FCarrinho); //195
end;
```

Nota do DevMan

SOLID é um acrônimo de acrônimos e reúne cinco siglas, cada uma com seu significado e importância. São cinco princípios fundamentais de POO que devem ser seguidos à risca para o desenvolvimento de um projeto de sucesso e fácil de se reusar, manter e estender.

SRP: Single Responsibility Principle (Princípio da Responsabilidade Única). Um objeto deve possuir uma única responsabilidade. Nada de classes faz-tudo, uma classe deve ter uma única responsabilidade e consequentemente um único motivo para mudar.

OCP: Open/Closed Principle (Princípio Aberto / Fechado). O software deve ser aberto para extensão, mas fechado para modificação. Se você precisa modificar o sistema inteiro por causa de um campo adicionado em uma tabela, então está tudo errado.

LSP: Liskov Substitution Principle (Princípio da Substituição de Liskov). Objetos devem ser substituídos por instâncias de seus subtipos (ou por instâncias de objetos que implementem as mesmas interfaces), sem afetar a correção do programa. O programa deve compilar normalmente e testes unitários não devem falhar caso uma substituição dessas seja feita.

ISP: Interface Segregation Principle (Princípio de Segregação da Interface). O mesmo que SRP mas aplicado a interfaces. Diz que é melhor ter muitas interfaces específicas do que uma "Geral".

DIP: Dependency Inversion Principle (Princípio de Inversão da Dependência). Deve-se depender apenas de interfaces abstratas e não de classes concretas, pois a implementação das classes deve ser livre para ser alterada sem que se quebre a interface e sem prejudicar o funcionamento do sistema.

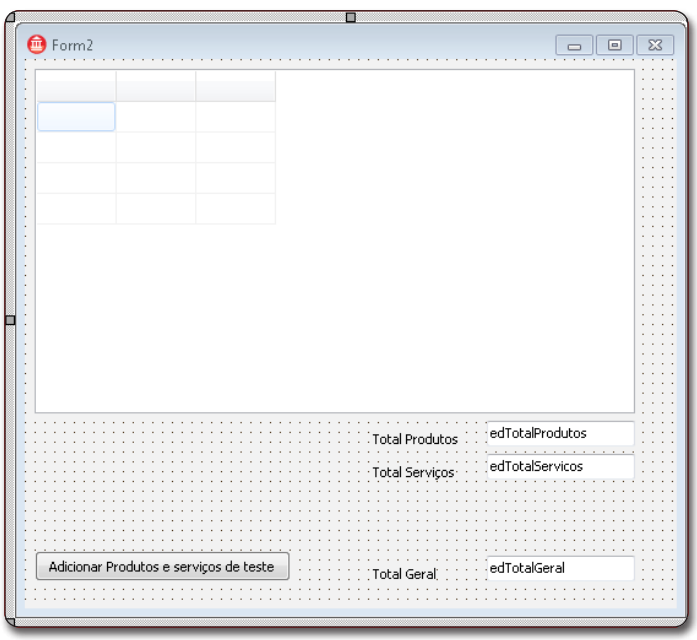


Figura 2. Pedido sem interfaces

A propriedade Itens da classe TCarrinho na **Listagem 8** merece uma atenção especial: Ela é uma propriedade indexada, assim como a propriedade Strings de um TStringList, a propriedade Fields de um dataset e outras propriedades que são comuns da VCL: Items, Columns, Values etc. Isso é possível declarando uma propriedade que tenha um argumento do tipo inteiro entre colchetes na sua declaração. Deve ser ligada a métodos acessores Get e Set um pouco diferentes: o método Set deve ter dois parâmetros, o primeiro deve ser o mesmo que estiver entre colchetes e o segundo deve ser um parâmetro do tipo da propriedade propriamente dita. O método Get além de retornar um objeto do mesmo tipo da propriedade recebe um parâmetro do tipo inteiro, o índice, da mesma forma que o Set. Esse tipo de propriedade serve para dois propósitos: primeiro o carrinho pode ser usado e varrido como um vetor, e segundo para expor a propriedade Items de sua Flista interna.

Nota

Embora possam existir interfaces com parâmetros genéricos (ou parâmetro "tipo") no escopo da própria interface (interfaces genéricas), as interfaces não podem ter métodos genéricos com parâmetros próprios.

Nota

Metaclasses são "Tipos Classe", ou seja, um tipo que é uma referência a uma classe. Uma variável do tipo metaclasse não representa uma instância desta classe, mas representa a classe em si, podendo ser usada para criar objetos da mesma. Como exemplo, o tipo TClass é um class of TObject. Quando se tem uma variável do A do tipo TClass, A não é uma instância de TObject, A é a própria classe TObject. Metaclasses são muito úteis para se usar em RTTI ou no lugar de generics onde eles não são possíveis. Da mesma forma que interfaces e generics as metaclasses fornecem uma forma de abstração e programação genérica.

Nota

Pode-se dizer que no desenvolvimento de software não existe certo ou errado, mas sim o que funciona e o que não funciona. É justamente para isso que existem as chamadas boas práticas da POO, para saber se o software está ou não de fato orientado a objetos, se ele é "manutenível" ou não e assim por diante.

Embora as classes TItemVenda e TCarrinho tenham sido implementadas de maneira interessante e parecerem estar corretas elas podem apresentar problemas futuramente. O objetivo deste artigo é expor os problemas dessa abordagem.

O primeiro problema que pode ser percebido é o fato de se ter uma classe que sirva para representar produtos e serviços. Isso foi feito usando uma propriedade Tipo ligada ao campo FTipo que é um enum TTipoItem = (tpProduto, tpServico). Quando se usa enums, força-se o uso de ifs e cases cada vez que a propriedade em questão deve ser verificada para saber de que tipo é.

Isto fere o princípio S.O.L.I.D. da responsabilidade única. Uma classe deveria ser produto ou serviço, e nada mais.

Esse problema já mostra uma consequência indesejada: a existência de um IF para calcular o subtotal deste item, no método GetTotal da classe TItemVenda.

Outra consequência são os memory leaks, ou vazamentos de memória. No evento OnCreate do formulário é necessário criar uma instância do Carrinho, e no evento OnDestroy é necessário destruir o carrinho e qualquer instância de ItemVenda que ele contenha. O uso de TObjectList em vez de TList minimiza esse problema, pois TObjectList destrói todos os seus objetos quando ele é destruído. Mesmo assim o TObjectList deve ser destruído.

A terceira consequência é o engessamento do Carrinho. Para adicionar regras de cálculo de impostos, técnicos e horas técnicas e obter o subtotal de produtos e serviços, deve-se varrer a coleção

de itens, perguntar seu tipo com um IF e aplicar os cálculos. Isso faria com que o IF do método GetTotal, que simplesmente verifica o tipo, fosse repetido de 1 a 4 vezes: uma vez se tudo fosse calculado em um único método, o que cria um método "macarrônico" muito longo e o uso de muitos campos privados representando Estado do objeto sendo usados como se fossem variáveis globais. Se fossem usados quatro métodos, um para obter o total de produtos, outro para obter o total de serviços, ICMS e ISS então seria pior, pois o IF seria copiado quatro vezes.

Para se obter o total de produtos e o total de serviços um código como o da **Listagem 10** deveria ser adicionado.

Listagem 10. Totalizando produtos e serviços

```
function TCarrinho.GetTotalProdutos: double;
var
 i: TItemVenda;
begin
 Result := 0;
 for i in FLista do
 begin
  if i.Tipo = tpProduto then
    Result := Result + i.GetTotal;
 end;
end;

function TCarrinho.GetTotalServicos: double;
var
 i: TItemVenda;
begin
 Result := 0;
 for i in FLista do
 begin
  if i.Tipo = tpServico then
    Result := Result + i.GetTotal;
 end;
end;
```

Considerando agora o mesmo software elaborado com interfaces e boas práticas da POO. A **Listagem 11** apresenta a interface de um Item de Venda. Um item de venda é qualquer coisa que pode ser "vendida" em uma empresa, é uma "linha" de uma nota fiscal ou orçamento. Ele pode representar um produto, serviço, aluguel de equipamento e assim por diante.

A interface serve para extrair e expor o que qualquer tipo de item tem: um subtotal. Analogamente ao exemplo anterior, um Item de Venda concreto como um Item de Produto deveria estar ligado a um produto e um Item de Serviço a um serviço. Como produto e serviço tem em comum o valor unitário e a descrição, eles poderiam implementar uma interface com essas informações. Por questão de simplicidade, o exemplo do artigo implementa apenas os itens de produto ou serviço, mas não os produtos ou serviços em si, com os dados de descrição, por exemplo.

Listagem 11. Interface IItemVenda

```
IItemVenda = interface
 ['{DEBCC45E-A4D3-4E47-B29C-80BDE802B375}']

  procedure SetValorUnitario(const valor: Double);
  function GetValorUnitario: Double;

  function GetTotal: Double;

  property ValorUnitario: Double read GetValorUnitario write SetValorUnitario;

end;
```

A propriedade ValorUnitario foi inserida nesta interface apenas para demonstrar que propriedades são possíveis em interfaces, contanto que elas estejam ligadas a métodos. Também é possível criar uma propriedade Total somente leitura.

Da mesma forma o carrinho pode ser implementado através de uma interface, e pedidos ou orçamentos podem implementar esta interface ou delegar sua implementação a um carrinho "concreto".

Tendo em mente que pedidos, orçamentos e Carrinho, todos, implementarão a interface ICarrinho é necessário pensar agora no como isso será feito, pois pode haver mais de uma maneira possível de se implementar a interface ICarrinho.

No que diz respeito à separação entre produtos e serviços podem haver duas listas, uma para cada tipo, ou usar RTTI para descobrir de que tipo o item é, na hora de obter o total de produtos e serviços.

As duas abordagens criam uma dependência entre o carrinho concreto e as implementações concretas de IItemVenda.

Uma terceira opção seria adicionar métodos ValorComoServico e ValorComoProduto à interface IItemVenda e suas implementações, mas isso induziria a três problemas:

- Necessidade de ValorComoProduto ser zero em serviços e vice-versa;
- Possibilidade de um objeto ter os dois valores por erro do programador;
- Além da "feiúra" do código resultante, um novo tipo de item como "aluguel" por exemplo teria de alterar a interface. Isso fere outro dos princípios S.O.L.I.D. que é o princípio Open/Closed. Isso significa que interfaces devem ser expandidas por meio de heranças e novas implementações, mas nunca alteradas.

Uma quarta forma, mais elegante, poderia ser Baseada em introduzir métodos genéricos que usem RTTI para verificar se determinado IITemVenda é de determinada classe e assim calcular os totais sem usar múltiplas listas e sem criar dependências entre as implementações concretas de Carrinho e Item de Venda.

Porém não é permitido criar métodos genéricos em interfaces, então contornamos essa limitação do Delphi criando métodos que recebam um parâmetro comum em vez de um parâmetro genérico. Esse parâmetro comum é uma metaclasse (T: TClass) onde TClass é definido na unit classes como TClass = class of TObject. A interface ICarrinho foi implementada com metaclasses e pode ser vista na **Listagem 12**.

Listagem 12. Interface ICarrinho

```
type
 ICarrinho = interface
  ['{BEF10108-F26B-4C0C-927D-C14CD65C7717}']

  procedure Adicionar(const item: IITemVenda);
  procedure Remover(const item: IITemVenda);
  function GetTotal: Double;  overload;
  function GetQuantidadeItens: Integer; overload;
  function GetTotal(T: TClass): Double;  overload;
  function GetQuantidadeItens(T: TClass): Integer;  overload;
  function GetItem(idx: integer): IITemVenda;
  property Itens[idx: integer]: IITemVenda read GetItem;

 end;
```

A classe TCarrinho, implementada na **Listagem 13**, tem apenas uma lista de itens e métodos para adicionar ou remover. Também possui métodos sobrecarregados para cálculo dos subtotais que aceitam uma classe como parâmetro. Isso permite que sejam calculados os totais gerais ou somente o total de produtos ou serviços.

Se for necessário adicionar uma classe TAluguel como item de venda, ou um milhão de classes diferentes, ainda assim se teria apenas dois métodos para calcular os totais.

A **Listagem 14** não é muito diferente da **Listagem 9**, mostra como tudo isso é usado. Uma das principais diferenças é que não é necessário destruir o carrinho no final, pois ele foi criado como uma interface.

Na **Listagem 14** foi necessário incluir um IF para verificar se o item é do tipo produto ou não para mostrar a quantidade. Esse IF seria desnecessário se criássemos uma classe de apresentação diferente para cada tipo de item ao apresentar/popular no string grid.

Como mostrado na **Figura 3** foram adicionados campos de impostos ISS e ICMS no formulário. Não vamos entrar em detalhes sobre o cálculo desses impostos..

A inclusão dos impostos é muito fácil. Basta acrescentar o método function GetImposto: Double; na interface IItemVenda e implementá-lo como sendo GetTotal * 0.05 na classe de serviços ou GetTotal * 0.18 na classe de produtos. Na interface ICarrinho é necessário colocar os métodos:

```
function GetTotalImpostos: Double; overload;
function GetTotalImpostos(T: TClass): Double; overload;
```

que seguem a mesma lógica dos métodos GetTotal e GetQuantidade. Ao se implementar esses métodos em TCarrinho eles já podem ser usados na GUI. O código é repetitivo, pois mudou apenas algumas linhas, por isso não será mostrado em uma nova listagem, mas o leitor pode conferir nos arquivos para download desta edição.

Implementar as classes TPedido e TOrcamento pode ser muito mais fácil agora. Essas classes apenas acrescentam funcionalidades ou informações que o TCarrinho não tem, talvez uma

Listagem 13. Classe TCarrinho

```
type
 TCarrinho = class(TInterfacedObject, ICarrinho)
 private
  FLista: TList<IITemVenda>;
 public
  constructor Create;
  destructor Destroy; override;

  procedure Adicionar(const item: IITemVenda);
  procedure Remover(const item: IITemVenda);
  function GetTotal: Double;  overload;
  function GetQuantidadeItens: Integer; overload;
  function GetTotal(T: TClass): Double;  overload;
  function GetQuantidadeItens(T: TClass): Integer;  overload;
  function GetItem(idx: integer): IITemVenda;
  property Itens[idx: integer]: IITemVenda read GetItem;

 end;
```

Listagem 14. Uso da classe TCarrinho

```
procedure TfrmInterfaces.Apresenta(carrinho: ICarrinho);
var i: integer;
begin

 StringGrid1.RowCount := carrinho.GetQuantidadeItens + 1; //o +1 é o cabeçalho
 for i := 0 to carrinho.GetQuantidadeItens -1 do
 begin
  StringGrid1.Cells[0, i+ 1] := FloatToStr( carrinho.Itens[i].ValorUnitario );
  if (carrinho.Itens[i] is TItemProduto) then
   StringGrid1.Cells[1, i+ 1] := FloatToStr((carrinho.Itens[i] as TItemProduto).
   Quantidade );
  StringGrid1.Cells[2, i+ 1] := FloatToStr( carrinho.Itens[i].GetTotal );
 end;

 edTotalGeral.Text := FloatToStr( carrinho.GetTotal );
 edTotalProdutos.Text := FloatToStr( carrinho.GetTotal(TItemProduto) );
 edTotalServicos.Text := FloatToStr( carrinho.GetTotal(TITemServico) );

end;

procedure TfrmInterfaces.btAdicionarTesteClick(Sender: TObject);
begin
 FCarrinho.Adicionar( TItemProduto.Create(15, 2) ); //30
 FCarrinho.Adicionar( TItemProduto.Create(13, 5) ); //65
 FCarrinho.Adicionar( TITemServico.Create(100) );   //100

 Apresenta(FCarrinho); //195
end;

procedure TfrmInterfaces.FormCreate(Sender: TObject);
begin
 StringGrid1.Cells[0, 0] := 'Unitario';
 StringGrid1.Cells[1, 0] := 'Quantidade';
 StringGrid1.Cells[2, 0] := 'Sub Total';
 FCarrinho := TCarrinho.Create;
end;
```

outra classe mande esses dados (pedido ou orçamento) para o banco de dados através de um ClientDataset ou uma camada de persistência e talvez um Pedido debite os itens do estoque enquanto o Orcamento não.

A principal ideia até aqui é mostrar que, o que foi feito até agora poderia ser feito com classes abstratas em vez de interfaces, isto porque ainda não foi empregado nenhum recurso exclusivo das interfaces. Mas o que vai ser feito agora não pode ser feito com classes abstratas, apenas com interfaces: a delegação.

Pedido, orçamento e nota fiscal tem em comum várias informações: a lista dos itens, os totais e subtotais, os cálculos dos impostos e assim por diante. Em vez de um pedido, orçamento ou nota fiscal herdar de TCarrinho e reimplementar/aproveitar/sobrescrever seus métodos, essas novas classes podem apenas "ter" um TCarrinho que implemente ICarrinho.

Formulários também podem implementar ICarrinho por delegação para apresentar as informações de um Carrinho.

A **Listagem 15** mostra como as três classes podem ser implementadas. Está sendo mostrado apenas a implementação básica de pedido, orçamento ou nota fiscal, sem suas propriedades ou métodos característicos. Uma coisa interessante que já é possível perceber aqui: não é necessário o cálculo dos impostos em Carrinho, Pedido ou Orçamento, apenas em NotaFiscal, sendo assim esses métodos de cálculo do imposto podem ser movidos de Carrinho para NotaFiscal e retirados da interface ICarrinho.

Como neste exemplo não foram implementados os métodos e propriedades específicos de TNotaFiscal, TPedido e

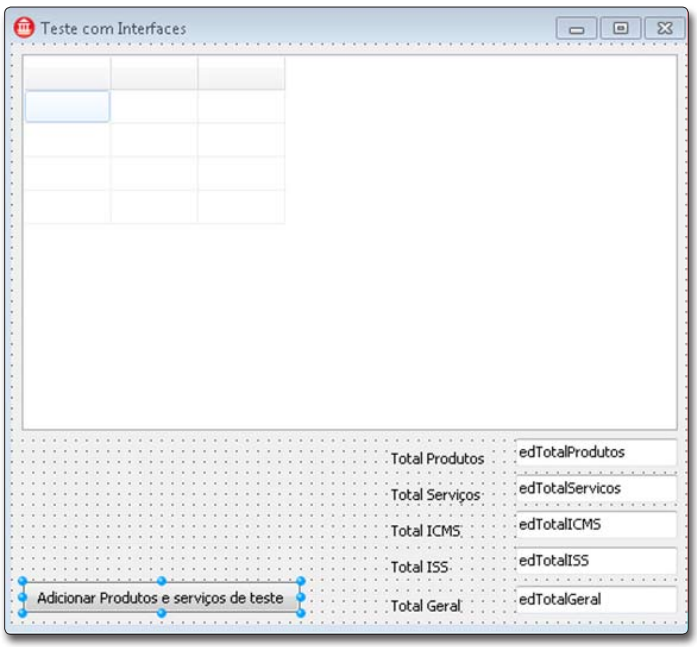


Figura 3. Janela de pedido com impostos

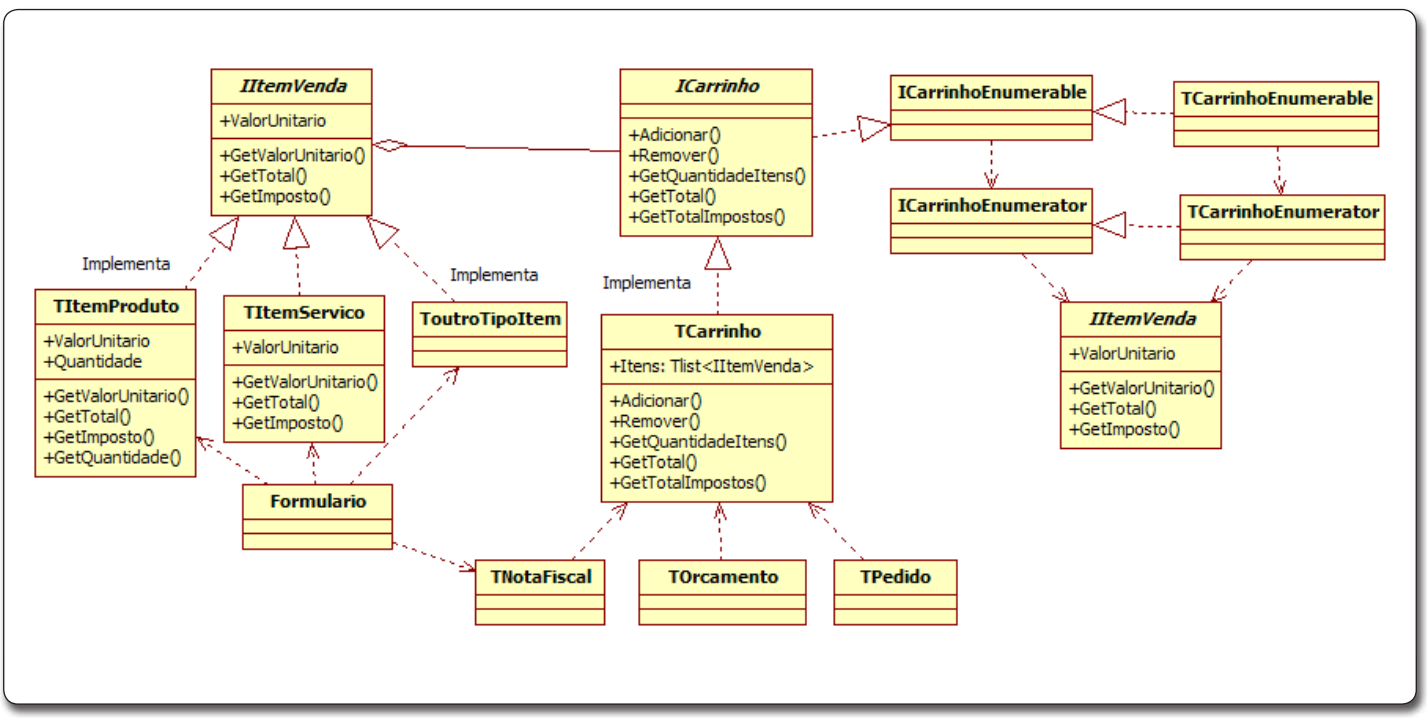


Figura 4. Diagrama de classes da solução

TOrcamento essas três classes são usadas da mesma maneira que TCarrinho. Essas classes implementam todos os métodos da interface ICarrinho sem necessariamente ser descendente de TCarrinho, e sem implementar ICarrinho diretamente. Basta que elas contenham um objeto qualquer que implemente a interface ICarrinho, isso é delegação.

Se neste exemplo tivesse uma classe TCarrinhoEspecial ou TCesta que também implementasse ICarrinho, um objeto desta classe poderia ser adicionado à TNotaFiscal para que essa implementasse as funcionalidades de ICarrinho. Isso tudo é possível sem que TCarrinhoEspecial seja descendente de TCarrinho e sem acrescentar um único método à TNotaFiscal, para ilustrar algo similar, a **Listagem 16** mostra como uma TNotaFiscal pode ser usada como ICarrinho (contanto que possua um em sua criação) e como ela pode ser usada como um ICarrinho, embora não tenha os métodos Adicionar, Remover etc... de ICarrinho.

Todo método que só aceitaria um ICarrinho como parâmetro pode aceitar uma TNotaFiscal, além disso os métodos de ICarrinho são acessíveis via RTTI com o operador "AS" fazendo typecast, por exemplo, como a seguir: (NotaFiscal as ICarrinho).Metodo.

Sempre que uma TNotaFiscal for usada como ICarrinho ela está usando na verdade o TCarrinho que existe dentro dela.

Além de exibir os subtotais de cada imposto, o campo FCarrinho do tipo ICarrinho foi trocado por um NotaFiscal do tipo TNotaFiscal. Ao se trabalhar com delegações, objetos agregados e objetos contidos trabalhe sempre com o objeto

Listagem 15. Classes TPedido, TOrcamento e TNotaFiscal

```
type
 TPedido = class(TInterfacedObject, ICarrinho)
 private
  FCarrinho: ICarrinho;
 public
  constructor Create(c: ICarrinho);
  property Carrinho: ICarrinho read FCarrinho implements ICarrinho;
 end;

 TOrcamento = class(TInterfacedObject,ICarrinho)
 private
  FCarrinho: ICarrinho;
 public
  constructor Create(c: ICarrinho);
  property Carrinho: ICarrinho read FCarrinho implements ICarrinho;
 end;

 TNotaFiscal = class(TInterfacedObject,ICarrinho)
 private
  FCarrinho: ICarrinho;
 public
  constructor Create(c: ICarrinho);
  property Carrinho: ICarrinho read FCarrinho implements ICarrinho;
 end;
```

Listagem 16. Uso da classe TNotaFiscal

```
type
 TfrmInterfaces = class(TForm)
  //o código abreviado, mas completo para download
  procedure FormCreate(Sender: TObject);
  procedure btAdicionarTesteClick(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
 private
  { Private declarations }
  NotaFiscal: TNotaFiscal;
 public
  { Public declarations }
  procedure Apresenta(carrinho: ICarrinho);
  procedure Adicionar(carrinho: ICarrinho;  item: IITemVenda);
 end;
procedure TfrmInterfaces.Adicionar(carrinho: ICarrinho; item: IITemVenda);
begin
 carrinho.Adicionar(item);
end;
procedure TfrmInterfaces.Apresenta(carrinho: ICarrinho);
var i: integer;
begin
 StringGrid1.RowCount := carrinho.GetQuantidadeItens + 1; //o +1 é o cabeçalho
 for i := 0 to carrinho.GetQuantidadeItens -1 do
 begin
  StringGrid1.Cells[0, i+ 1] := FloatToStr( carrinho.Itens[i].ValorUnitario );
  StringGrid1.Cells[2, i+ 1] := FloatToStr( carrinho.Itens[i].GetTotal );

  if (carrinho.Itens[i] is TItemProduto) then
  begin
   StringGrid1.Cells[1, i+ 1] := FloatToStr((carrinho.Itens[i] as TItemProduto).Quantidade );
   StringGrid1.Cells[3, i+ 1] := FloatToStr( carrinho.Itens[i].GetImposto );
  end else if (carrinho.Itens[i] is TITemServico) then
  begin
   StringGrid1.Cells[4, i+ 1] := FloatToStr( carrinho.Itens[i].GetImposto );
  end;
 end;

 edTotalGeral.Text := FloatToStr( carrinho.GetTotal );
 edTotalProdutos.Text := FloatToStr( carrinho.GetTotal(TItemProduto) );
 edTotalServicos.Text := FloatToStr( carrinho.GetTotal(TITemServico) );
 edTotalICMS.Text := FloatToStr( carrinho.GetTotalImpostos(TItemProduto) );
 edTotalISS.Text := FloatToStr( carrinho.GetTotalImpostos(TITemServico) )
end;

procedure TfrmInterfaces.btAdicionarTesteClick(Sender: TObject);
begin
 Adicionar(NotaFiscal, TItemProduto.Create(15, 2) ); //30
 Adicionar(NotaFiscal, TItemProduto.Create(13, 5) ); //65
 Adicionar(NotaFiscal, TITemServico.Create(100) );   //100
 Apresenta(NotaFiscal); //195
end;

procedure TfrmInterfaces.FormCreate(Sender: TObject);
begin
 StringGrid1.Cells[0, 0] := 'Unitario';
 StringGrid1.Cells[1, 0] := 'Quantidade';
 StringGrid1.Cells[2, 0] := 'Sub Total';
 StringGrid1.Cells[3, 0] := 'ICMS';
 StringGrid1.Cells[4, 0] := 'ISS';
 NotaFiscal := TNotaFiscal.Create( TCarrinho.Create);
end;
```

propriamente dito e não com a interface. Isso se dá porque esses tipos de objetos tem uma maneira diferente de contar as referências. As interfaces contam suas referências, limpando aquelas que não estão mais em uso. Usar os objetos manteria viva uma referência, impedindo que a mesma seja liberada da memória. Se não for utilizada dessa forma um objeto com uma interface delegada pode ser liberado da memória tendo o objeto que ele contém, sendo liberado também antes de ser desnecessário.

Por isso, a nota fiscal é criada com NotaFiscal := TNotaFiscal.Create(TCarrinho.Create) e destruída no evento OnDestroy do formulário com NotaFiscal.Free.

Além disso, foi incluído o método Adicionar, que recebe um objeto que implemente ICarrinho, como NotaFiscal, e um item e adiciona o item ao carrinho. Esse método é meramente ilustrativo, serve para mostrar como uma NotaFiscal pode "se passar" por carrinho.

Pode-se imaginar, a partir desse ponto, como seria a inclusão de técnicos e horas técnicas para o cálculo do valor do serviço. Um TItemServico deveria ter uma coleção de técnicos, cada qual com o número de horas trabalhadas, mas deixaremos esse exercício a cargo do leitor.

A **Figura 4** mostra o modelo completo. Repare que não há dependência entre as classes concretas que implementam ICarrinho e as classes concretas que implementam IItemVenda. A seta tracejada de ponta sólida representa a "realização", ou implementação de uma interface. A seta tracejada de ponta aberta (pé de galinha) representa dependência. Existe dependência entre TNotaFiscal e TCarrinho, por exemplo, pois em TNotaFiscal é um objeto do tipo TCarrinho que implementa a interface ICarrinho para ela.

O diagrama não está completo, não mostra todos os métodos e propriedades de todas as classes e nem os acessores do tipo Set (modificadores). O objetivo desse diagrama de classes é fornecer um esboço de como as classes são ligadas entre si, quais são as dependências e que classe implementa cada interface.

Interfaces importantes da VCL e RTTI

Há interfaces no Delphi que devem ser implementadas para que certos comportamentos sejam conseguidos. Existem muitas interfaces especiais no Delphi e não é objetivo desse artigo exemplificar todas. Existem interfaces para comunicação com o próprio IDE e criação de plugins, existem interfaces que implementadas originam objetos que podem ser serializados. O ClientDataset e o DatasetProvider implementam muitas interfaces especiais para lidar com dados.

Um recurso interessante do Delphi é o loop for...in. Ele varre uma coleção sem usar um índice inteiro, mas atribuindo o item atual a uma variável. Sua interpretação é: "para cada item nesta coleção faça" e algumas linguagens chamam esse operador de foreach.

Esse operador não funciona em qualquer coleção. Em todas as coleções nativas da VCL, inclusive coleções primitivas como Arrays e antigas como TObjectList (não genérico) o pessoal da Embarcadero teve o cuidado de tornar compatíveis com o for...in, entretanto coleções criadas pelo programador que não são derivadas das coleções da VCL não são compatíveis.

O mesmo acontece com classes que se comportam como coleções. Por exemplo, TNotaFiscal e TCarrinho contém uma coleção, mas não são uma. Esse é um comportamento típico de classes que servem de wrappers para encapsular coleções. Foi criada a propriedade Itens na interface ICarrinho que se implementada faz com que as classes como TCarrinho, TNotaFiscal etc se comportem como coleções.

Nota

propriedade .GetValue não funcionará corretamente se for passado inst como parâmetro, já que inst é uma instância "nova". Para isso, use (item as TObject).

Nota

No Delphi XE2 há um método chamado GetDeclaredImplementedInterfaces e outro GetImplementedInterfaces que serve para, dada uma instância, listar todas as interfaces que ela implementa em sua declaração e todas as interfaces que ela implementa ao longo de sua linhagem, respectivamente.

As informações mostradas a partir deste ponto são especialmente úteis se você tiver algum sistema legado feito em uma versão mais antiga do Delphi quando era necessário ter classes derivadas de TObjectList (da unit Contnrs) porque ainda não existiam as listas genéricas. Com uma alteração de poucas linhas é possível definir enumerators para listas antigas de objetos e componentes sem a necessidade de se alterar muito código.

Para que TNotaFiscal seja usada com for...in, ela deve implementar a interface IEnumerable e ter um objeto GetEnumerator que retorna um INumerator. Entretanto, IEnumerable e INumerator trabalham apenas com objetos TObject e não com interfaces, e nossas listas são de interfaces. Além disso, a interface genérica IEnumerable<T> não dispensa a implementação dos métodos básicos que lidam com TObject, causando assim um conflito de interfaces e o mesmo ocorre com IEnumerator<T>.

Para solucionar isso, foram criadas as interfaces ICarrinhoEnumerable e ICarrinhoEnumerator conforme mostrado na **Listagem 17** juntamente com a nova interface ICarrinho, onde foi adicionado um único método.

Como os enumerators são varridos via RTTI, não é necessário se preocupar com o fato de ICarrinhoEnumerator e ICarrinhoEnumerable não serem descendentes de IEnumerator e IEnumerable. Basta que os "enumerables" tenham um método chamado GetEnumerator e que os "Enumerators" tenham os métodos GetCurrent e MoveNext.

A melhor forma de implementar ICarrinhoEnumerable sem alterar muitas linhas da aplicação é fazer com que a interface ICarrinho seja descendente de ICarrinhoEnumerable e assim todas as classes que implementarem ICarrinho estarão implementando ICarrinhoEnumerable junto.

ICarrinhoEnumerable te obriga a adicionar o método GetEnumerator que retorna um objeto da interface ICarrinhoEnumerator. É necessário criar um Enumerator concreto para a classe TCarrinho.

Este mesmo enumerator concreto pode ser usado para outras classes que implementam ICarrinho, ou pode-se desenvolver outros. O importante é que tudo seja genérico o suficiente para não se ter excesso de classes de enumerators, tornando a manutenção difícil.

As classes TNotaFiscal, TPedido e TOrcamento, que delegam a implementação de ICarrinho a um objeto do tipo TCarrinho não precisam implementar ICarrinhoEnumerable, já que TCarrinho implementa.

A **Listagem 18** contém a interface TCarrinho com o novo método. Não possui a implementação de TCarrinho porque seria repetitivo. A única coisa que importa de TCarrinho é a implementação do método GetEnumerator que retorna Result := TCarrinhoEnumerator.Create(Self), ou seja, um objeto do tipo TCarrinhoEnumerator que recebe como parâmetro de criação o próprio carrinho.

Um Enumerator é simples, ele apenas usa um índice do tipo Integer (FIndice) para controlar qual item será retornado. Na criação e no método Reset esse índice é definido como -1 (valor padrão), o que significa que as listagens começam com zero e que -1 é o resultado para listas vazias.

Seu método GetCurrent retorna o IItemVenda corrente de índice FIndice. MoveNext é chamado a cada iteração do loop for...in enquanto estiver no loop e FIndice é incrementado a cada chamada. Ele não é incrementado se FIndice for igual ao último índice. Se isso acontecer MoveNext retorna false e o loop acaba.

Falando em RTTI, é possível usá-lo com interfaces. Até a versão XE do Delphi não é possível obter informações de RTTI de uma variável do tipo interface, apenas de um tipo "Objeto", mesmo que a instância seja a mesma. Isso não é problema, já que qualquer interface pode ser convertida para TObject ou TInterfacedObject com o operador AS.

Fazendo isso é possível obter as informações de tipo da classe de uma instância qualquer e todas as informações ligadas a ela. Foi adicionado um memo no formulário de pedido para listar informações do IITemVenda sendo adicionado para mostrar como é fácil obter informações de tipo de uma interface.

O método Adicionar foi alterado para mostrar no memo o nome da classe, nome dos métodos, nome das propriedades e valores de todos os itens adicionados. Esse método encontra-se na **Listagem 19**.

A variável ctx é do tipo TRTTIContext e serve para se varrer informações sobre os tipos. A variável typ representa um conjunto de informações de um tipo. A variável inst é do tipo TRTTIInstanceType e representa uma instância de um tipo. Ela pode ser obtida com typ.AsInstance.

Metodos é um array genérico de TRttiMethod. O mesmo vale para Propriedades.

O que há de novo aqui são os métodos GetDeclaredMethods e GetDeclaredProperties. Esses métodos retornam apenas as listas de métodos e propriedades declarados no escopo da classe, na seção interface da unit, e não todos os outros métodos e propriedades que a interface herda de seus ancestrais, bem como todos os métodos e propriedades de TObject.

Considerando que no teste são adicionados dois itens de produto e um item de serviço, e que os valores são, respectivamente dois produtos de R$15,00, cinco produtos de R$13,00 e um serviço de R$100,00, esses resultados são apropriadamente exibidos no memo como mostra a **Listagem 20**.

Listagem 17. ICarrinhoEnumerator e ICarrinhoEnumerable

```
ICarrinhoEnumerator = interface(IInterface)
  ['{8AF6FB49-8984-4342-99D4-BD52EB6374BF}']
  function GetCurrent: IITemVenda;
  function MoveNext: Boolean;
  procedure Reset;
  property Current: IITemVenda read GetCurrent;
end;

ICarrinhoEnumerable = interface(IInterface)
  function GetEnumerator: ICarrinhoEnumerator;
end;

ICarrinho = interface(ICarrinhoEnumerable)
  ['{BEF10108-F26B-4C0C-927D-C14CD65C7717}']
  procedure Adicionar(const item: IITemVenda);
  procedure Remover(const item: IITemVenda);
  function GetTotal: Double;  overload;
  function GetQuantidadeItens: Integer; overload;
  function GetTotal(T: TClass): Double;  overload;
  function GetQuantidadeItens(T: TClass): Integer;  overload;
  function GetTotalImpostos: Double; overload;
  function GetTotalImpostos(T: TClass): Double; overload;
  function GetItem(idx: integer): IITemVenda;
  function GetEnumerator: ICarrinhoEnumerator;
  property Itens[idx: integer]: IITemVenda read GetItem;
end;
```

Listagem 18. Implementando IEnumerable e IEnumerator

```
TCarrinho = class(TInterfacedObject, ICarrinho)
private
  FLista: TList<IITemVenda>;
public
  constructor Create;
  destructor Destroy; override;

  procedure Adicionar(const item: IITemVenda);
  procedure Remover(const item: IITemVenda);
  function GetTotal: Double;  overload;
  function GetQuantidadeItens: Integer; overload;
  function GetTotal(T: TClass): Double;  overload;
  function GetQuantidadeItens(T: TClass): Integer;  overload;
  function GetTotalImpostos: Double; overload;
  function GetTotalImpostos(T: TClass): Double; overload;
  function GetItem(idx: integer): IITemVenda;
  function GetEnumerator: ICarrinhoEnumerator;
  property Itens[idx: integer]: IITemVenda read GetItem;

end;
```

Conclusão

Programar guiado por interfaces é o começo para uma boa e sólida programação levando a um código robusto e de fácil manutenção. Mesmo que seu software não use interfaces

Listagem 19. Usando interfaces e RTTI

```
procedure TfrmInterfaces.Adicionar(carrinho: ICarrinho; item: IItemVenda);
var
  ctx : TRttiContext;
  typ: TRttiType;
  inst: TRttiInstanceType;
  Metodos: TArray<TRttiMethod>;
  metodo: TRttiMethod;
  Propriedades: TArray<TRttiProperty>;
  propriedade: TRttiProperty;
begin
  carrinho.Adicionar(item);

  ctx := TRttiContext.Create;
  typ := ctx.GetType((item as TObject).ClassType);
  inst := typ.AsInstance;
  Metodos := typ.GetDeclaredMethods;
  Propriedades := typ.GetDeclaredProperties;

  mPropriedades.Lines.Add(typ.Name);
  mPropriedades.Lines.Add('-----Metodos----');
  for metodo in Metodos do
    mPropriedades.Lines.Add(metodo.Name);
  mPropriedades.Lines.Add('-----Propriedades----');
  for propriedade in Propriedades do
    mPropriedades.Lines.Add(propriedade.Name + ' - ' + propriedade.GetValue
    ((item as TObject) ).ToString);
  mPropriedades.Lines.Add('==================');
end;
```

Listagem 20. Saída do método Adicionar para o Memo

```
TItemProduto
-----Metodos----
Create
SetValorUnitario
GetValorUnitario
GetTotal
GetImposto
-----Propriedades----
ValorUnitario - 15
Quantidade - 2
==================
TItemProduto
-----Metodos----
Create
SetValorUnitario
GetValorUnitario
GetTotal
GetImposto
-----Propriedades----
ValorUnitario - 13
Quantidade - 5
==================
TITemServico
-----Metodos----
Create
SetValorUnitario
GetValorUnitario
GetTotal
GetImposto
-----Propriedades----
ValorUnitario - 100
==================
```

"explicitamente" é necessário que ele as use "implicitamente" fazendo uma boa separação entre a interface dos objetos e sua implementação.

O uso de interfaces é um requisito mais do que necessário para um desenvolvimento de acordo com os princípios S.O.L.I.D., principalmente do que diz respeito à substituição de Liskov, pois interfaces permitem que objetos sejam trocados por outros que não tenham uma superclasse em comum.

A programação orientada por interfaces não surgiu com a programação orientada a objetos, muito pelo contrário: Units do antigo pascal e arquivos cabeçalho (.H) do C já exploravam o conceito de separação de interface e implementação para mudanças de implementação, múltiplas implementações, organização de código, evitar repetição e assim por diante. Porém com a POO as interfaces encontraram sua "morada" definitiva e as duas coisas evoluiram juntas.

Pode-se pensar que interface é a base da abstração, assim como abstração é o princípio do aprendizado. Quando a roda foi inventada logo se pode perceber que tudo que devesse "rolar" deveria ser ou ter uma roda. O mesmo se aplica com interfaces.

Não é possível em um único artigo explorar totalmente interfaces e ainda mostrar muitos exemplos práticos, mas o objetivo do artigo é dar ao leitor a certeza de que se ele tem classes de domínios diferentes mas com métodos em comum e que devem poder ser intercambiáveis então pode-se extrair uma interface dentre elas. Qualquer comportamento comum entre dois objetos pode ser extraído na forma de uma interface e até mesmo sequências intermináveis de ifs e cases podem ser minimizados com um bom uso de interfaces e princípios S.O.L.I.D.

O leitor já pode procurar onde criar ou implementar interfaces nos seus softwares e começar agora mesmo a fazer os devidos refactorings.

Vitor Luiz Rubio
vitorrubio@gmail.com - http://blog.vitorrubio.com.br/)
Tecnólogo em Processamento de Dados pela FATEC-SP e MCP. Programa com Delphi desde a versão 3 e com .Net desde a versão 1.1.

Delphi Reference Counted Interfaces
http://edn.embarcadero.com/article/30125

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Com quantas mentes se revoluciona o mundo?

Campus Party™

6 a 12 de fevereiro de 2012
Anhembi (SP)

Inovação
Cultura Digital
Ciência
Entretenimento Digital

Unimos talento · Criamos futuro

www.campus-party.com.br

@CampusPartyBRA
www.facebook.com/campuspartybrasil